# “Menuz – Conception, design and testing of a split menu for smartphone”

Dissertation presented by
**Nathan MAGROFUOCO ,**

for obtaining the Master’s degree in
**Management**

Supervisor(s)
**Jean VANDERDONCKT**

Reader(s)
**Charles PÊCHEUR, Sébastien COMBÉFIS**

Academic year 2016-2017

"It is far better to adapt the technology to the user than to force the user to adapt to the technology."

*Larry Marine*

# 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 and researches realized about menu usability. The objective is to understand how computer scientists are able to understand users in order to design usable menu interfaces. This state of the art is the starting point of the entire experiment. It allows to visualize the problem and to reveal the areas of improvement that can be provided for the topic.

## 1 Massive Open On-line Courses

### 1.1 Statistical analysis of the archive

Let's take a deeper look at this archive. Table 2.1 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 2.3a 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 same as the participation in the first exercise of LOUV1.2X. It seems that learners who

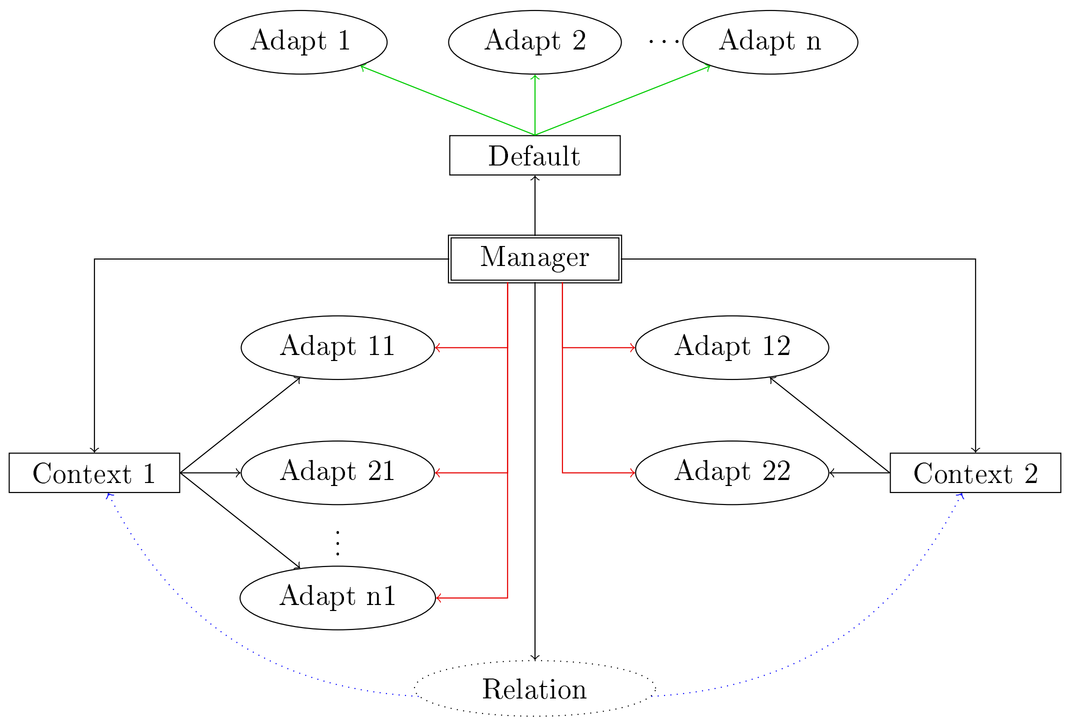


Figure 2.1: Architecture of our Context-Oriented framework

$\color{red}{\longrightarrow}$ Exists when the adaption context is activated

$\color{green}{\longrightarrow}$ Exists when at least one adaptation was activated for the function

$\cdots\cdots\rightarrow$ Link with a source and a destination context label

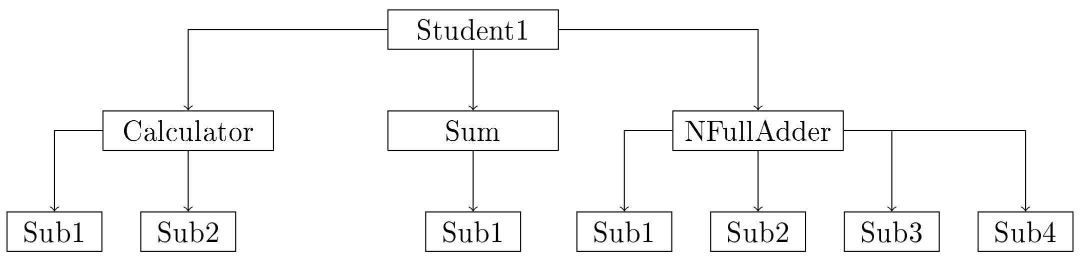
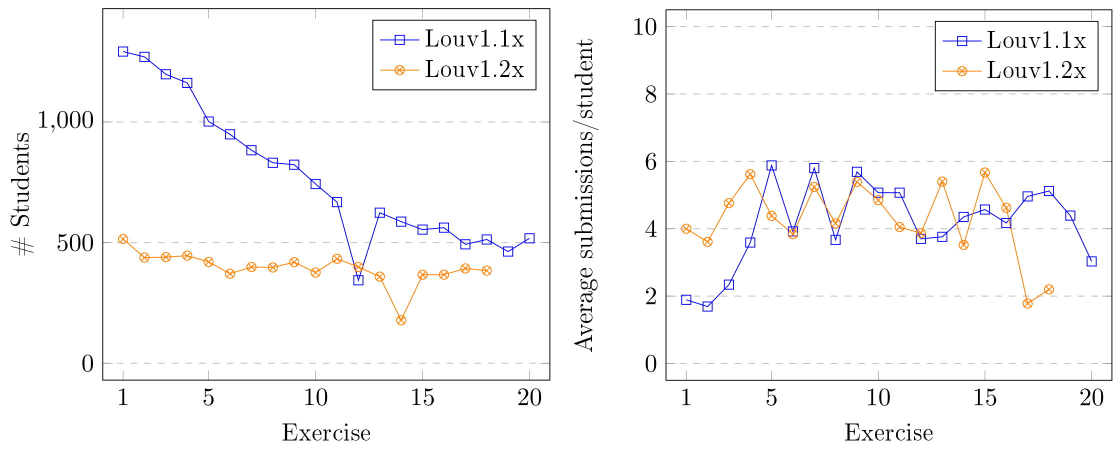


Figure 2.2: Folder organisation for each student

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 2.3a. 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 C.

| | |
|---|---|
| # 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 2.1: Recapitulative table concerning students, exercises and submissions



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

Figure 2.3

Fig 2.3b 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 2.4 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 2.4 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.

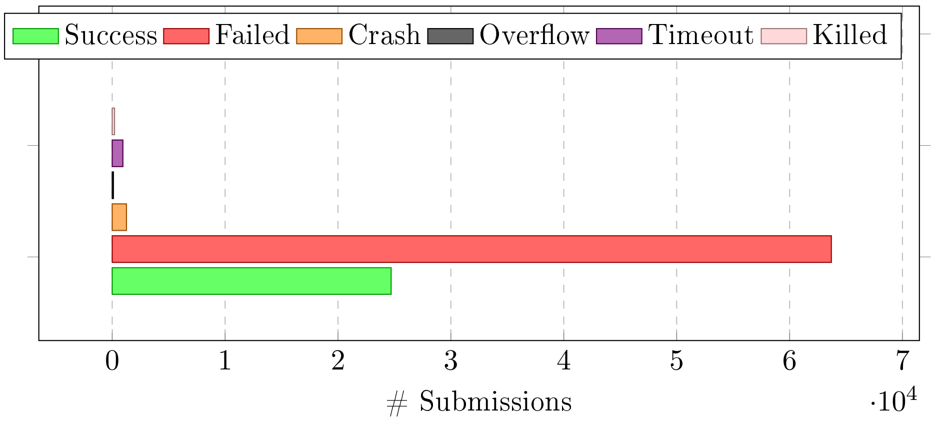


Figure 2.4: Count of submissions for each grade

The comparison between table 2.1 and fig 2.4 shows another interesting feature. As mentioned in table 2.1, 22,580 exercises have been tried by the students and exactly 24,709 submissions are considered as *successful* by INGInious as shown in figure 2.4. 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.

# 2 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.

## 2.1 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.

## 2.2 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 2.1. 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.

## 2.3 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.

### 2.3.1 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 2.5:

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 2.6.

```
fun {MainSum N}
  local Sum in
    fun {Sum N Acc}
      [YOUR CODE]
    end
    {Sum N 0}
  end
end
```

Figure 2.5: Pre-defined piece of code for the Sum exercise

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

Figure 2.6: 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 2.7. 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 2.7: First selected submission for the Sum exercise

```
if N==0 then Acc
else Sum N-1 Acc+N*N}
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 2.8 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 2.8: Second selected submission for the Sum exercise

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

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

Fig 2.9 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.

### 2.3.2 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.

```
    if N==1 then 1
    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.

```
    if N==0 then 1
    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.

```
    if N==0 then 1+Acc
    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.

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

Figure 2.9: Four successive submissions from the same student for the Sum exercise

## 2.4 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.

## 3 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.

### 3.1 Syntax errors markers

#### 3.1.1 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:

```
<var1> = <var2> = <value>     /* not allowed in Oz */

<var1> = <value>              /* should be used instead */
<var2> = <value>
```

### 3.1.2 If statements

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

```
if <condition> then <statement>
elseif <condition> then <statement>  /* optional */
else <statement>
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."**

### 3.1.3 Case statements

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

```
case <expression> of <pattern0> then <statement>
[<pattern1>] then <statement>     /* optional */
[<pattern2>] then <statement>     /* optional */
else <statement>
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."**

### 3.1.4 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."**

### 3.1.5 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."**

## 3.2 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."**

## 3.3 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.”**

## 4 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 3

# 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" [4]. 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 [5]. 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 3.1. 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 3.2.

```
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 3.1: Regular expressions for integers

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

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

Figure 3.2: 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 3.3. Unfortunately, the Oz grammar described in [5] 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 [6]. 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 [7].

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

Figure 3.3: Operators in EBNF

Fig 3.4 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 [5]. 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 3.4: Syntactic rule defining a nested declaration in EBNF notation

Fig 3.5 (resp. fig 3.6) provides a single-line part of the syntactic rule for a *nested condition* (resp. *expression*) as described in [5]. 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 [5], 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 3.5 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 A.

$$\text{NestCon::=} \quad \langle\text{expression}\rangle \quad (\text{'='} \mid \text{':='} \mid \text{','}) \quad \langle\text{expression}\rangle$$

Figure 3.5: Part of the syntactic rule for a nested condition

$$\langle\text{expression}\rangle \ ::= \quad \text{NestCon}$$

Figure 3.6: 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.

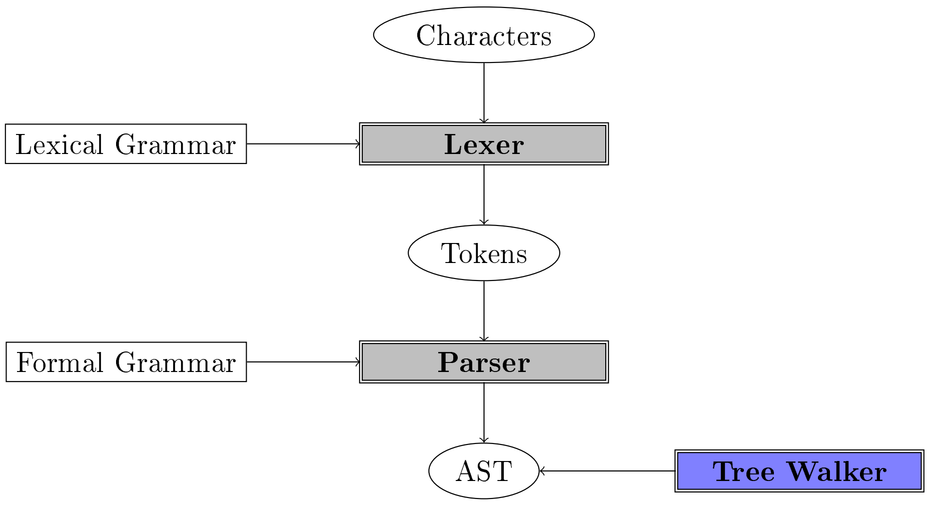


Figure 3.7: Lexer and parser workflow

**Lexical analysis** The lexer is in charge of the lexical analysis. As shown in fig 3.7, it corresponds to the action of splitting the input stream of characters into a sequence of corresponding tokens from the lexical grammar.

Fig 3.8 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 3.8: 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 3.7, the output of the parser is an AST, an *Abstract Syntax Tree* which is a tree representation of the parsed program.

Fig 3.8 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 3.9. "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 3.10.

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

Figure 3.9: 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 3.7. This tree walker supplies several classes and interfaces to visit the ASTs output by the parser. Fig 3.11 provides an overview of the

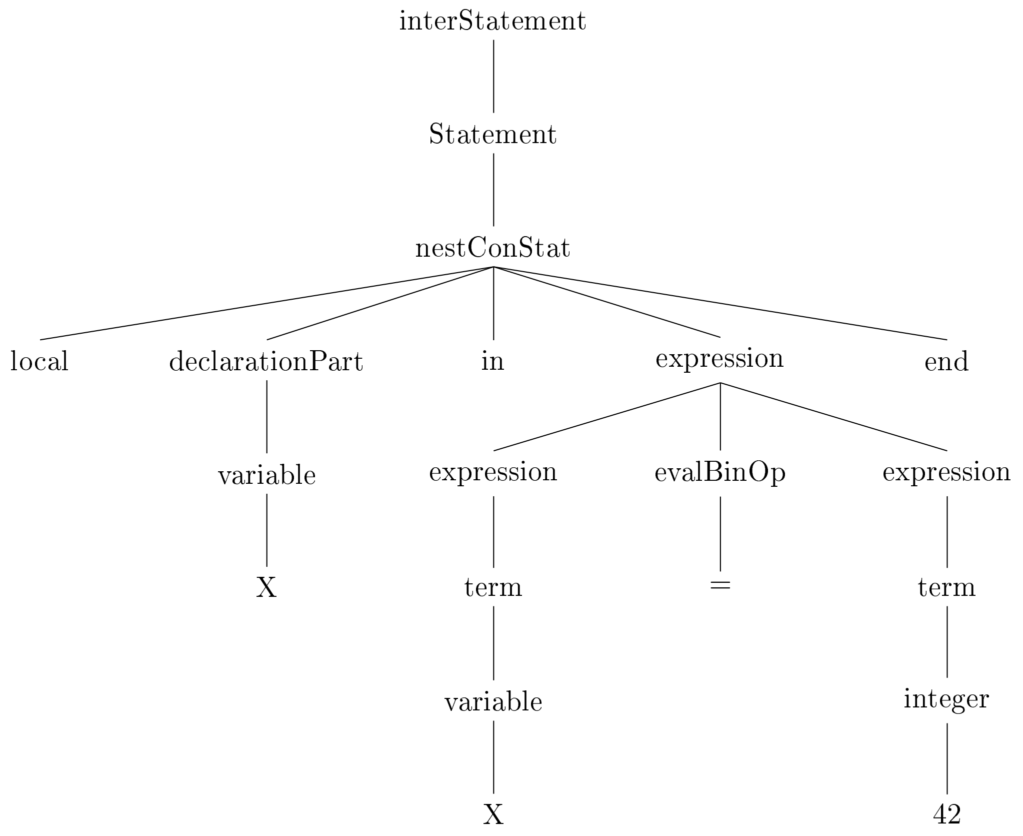


Figure 3.10: AST generated by the parser from the input at fig 3.8

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.

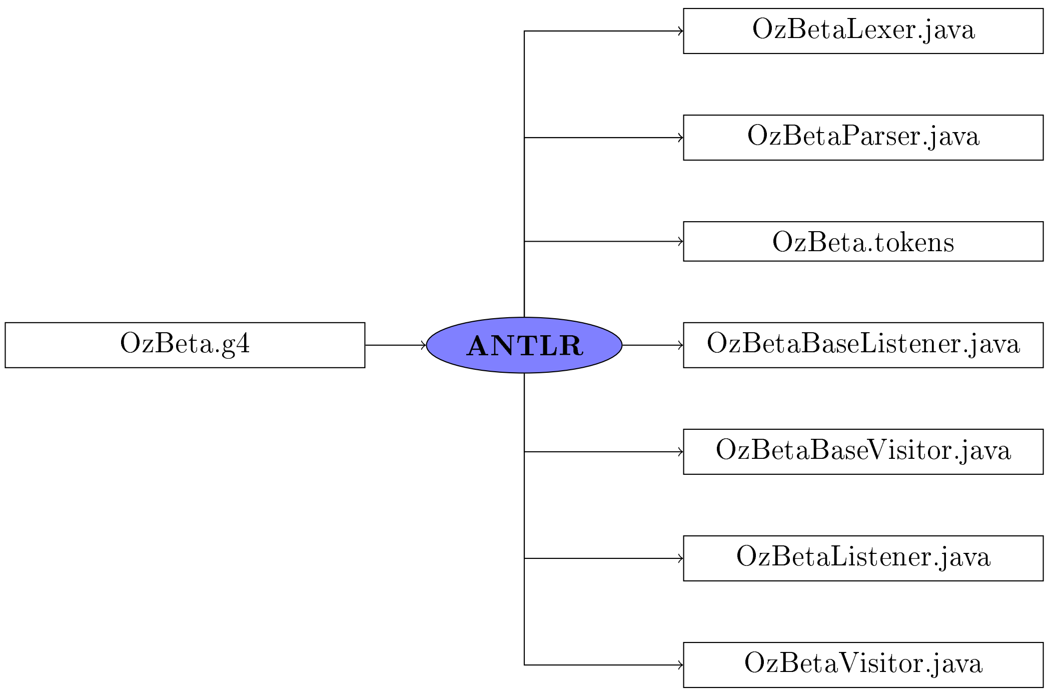


Figure 3.11: ANTLR generated files from the input grammar

**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 3.12 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 3.13 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 3.9. 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.

```
@Override public void enterNestDec(OzBetaParser.NestDecContext ctx) {
    if( ctx.getChild(0).getText().equals("proc") ||
        ctx.getChild(0).getText().equals("fun")) {

        String procName = ctx.getChild(2).getText();

        if(!ctx.getChild(1).getText().equals("{")) {
            procName = ctx.getChild(1).getText();
            CustomErrorListener.addError(ctx,
            OzInterpretor.vectorsArray.get(53));
        }

        else if(!ctx.getChild(ctx.getChildCount()-3).getText().equals
        ("}")) {
            CustomErrorListener.addError(ctx,
            OzInterpretor.vectorsArray.get(54));
        }
    }
}
```

Figure 3.12: Implementation of the nested declaration entering event

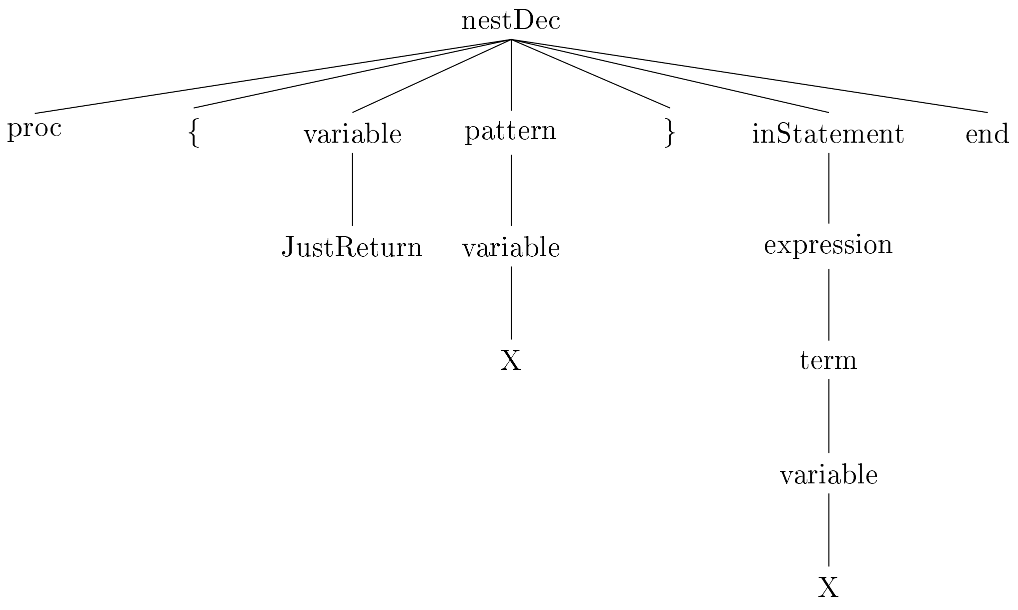


Figure 3.13: Tree representation of a potential NestDecContext

Fig 3.14 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 D.

- 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 3.14: 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 3.15 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 3.15 provides the complete feedback output by the *CustomErrorListener*.

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

```
String [] variables = {ctx.getChild(0).getText()};
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 3.15: Example of error reporting

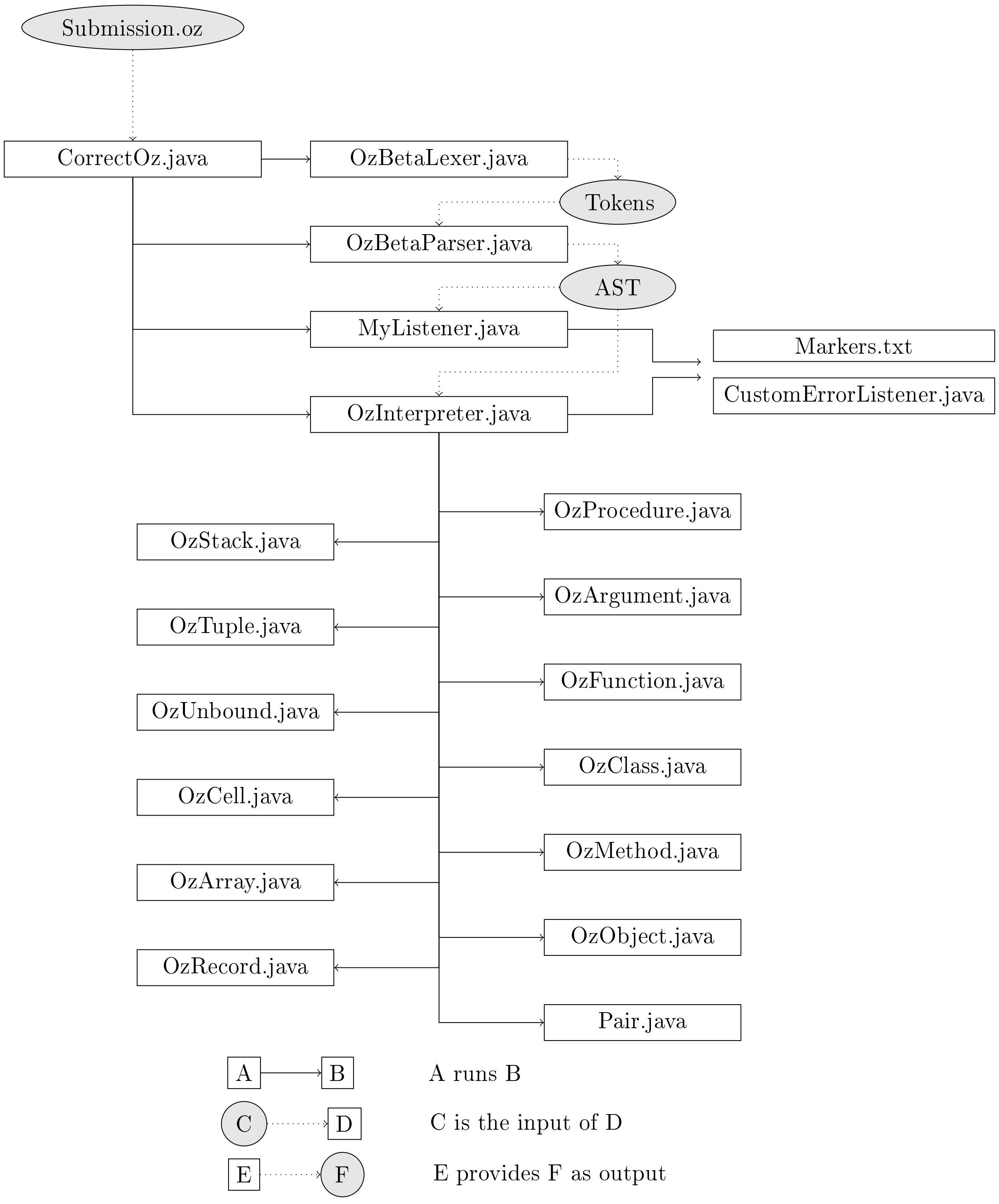


Figure 3.16: Architecture of the interpreter

## 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 3.16 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 **??**, section 3 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 3.17 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 3.17: Example of HashMap environment

**Store** The store is another *HashMap* data structure. It binds each store variable to its corresponding value. The triplet {identifier ↦ store variable ↦ value} is an essential concept in Oz. Fig 3.18 describes a potential store related to the environment provided in fig 3.17. 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.

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

Figure 3.18: Example of HashMap store related to fig 3.17

**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 3.19 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.

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

Figure 3.19: 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 3.20. 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 3.21 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 3.20: 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 3.21: 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 4

# 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 A .

**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" [8]. 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 4.1 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 4.1: Docker configuration file

containers. This section describes each file provided to INGInious in addition to the Docker configuration file. Fig 4.2 provides an overview of the interactions between those files.

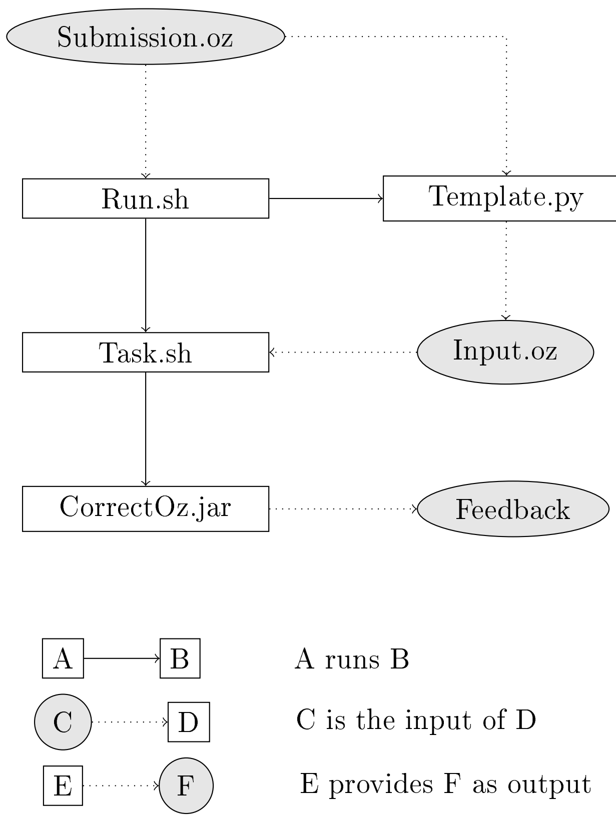


Figure 4.2: Interactions between the files provided to INGInious

**CorrectOz** is a ".jar" archive that groups the entire architecture presented in chapter 3, 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 3, 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 4.3 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.

```
local BrowseFormula in
   proc {BrowseFormula}
       @@BrowseX@@
   end
 {BrowseFormula}
end
```

Figure 4.3: 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 4.4 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 4.5 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 4.4: 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 4.5: 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 4.6 provides the command line stored in the *task* file to run correctly the interpreter and fig 4.7 provides the complete implementation of the *template* file.

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

Figure 4.6: Command line for exercise CalledOnlyOnce

```
local X SlowAdd in
  fun {SlowAdd X Y}
    {Delay 1000}
    X+Y
  end

  @@CalledOnlyOnce@@
end
```

Figure 4.7: 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 **??**, subsubsection 2.3.1 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 4.9 and the command line of the *task* file by fig 4.8.

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

Figure 4.8: Command line for exercise Sum

```
local MainSum R in
  fun {MainSum N}
    local Sum in
      fun {Sum N Acc}
        @@Sum@@
      end
      {Sum N 0}
    end
  end
  R = {MainSum 8}
end
```

Figure 4.9: 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 4.10 provides the complete command line run by the *task* file and fig 4.11 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 4.10: Command line for exercise Append

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

Figure 4.11: 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 4.12 at lines 11 and 18.

```python
# 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 4.12: 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 [9]. 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 5

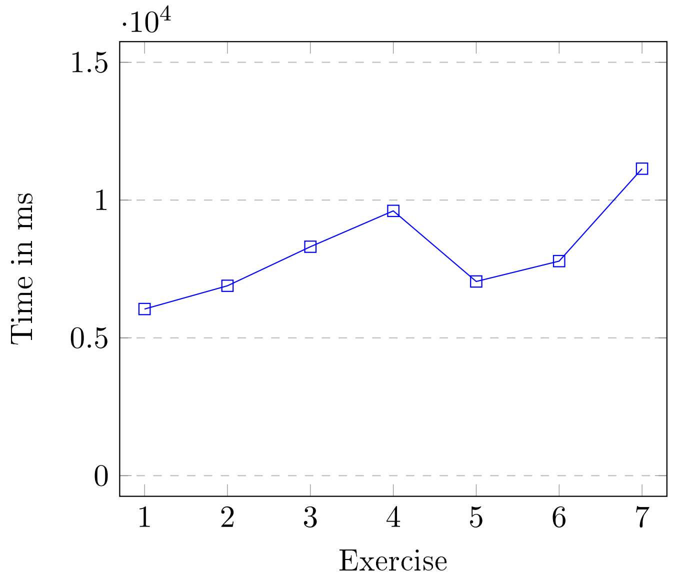
# Evaluation

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.

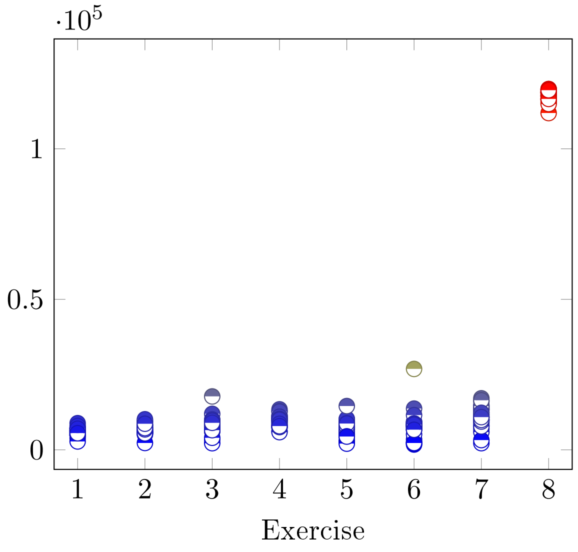
## 1 Execution time

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 5.1a 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 5.1b provides the complete results that allow to compute the average time described by fig 5.1a 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

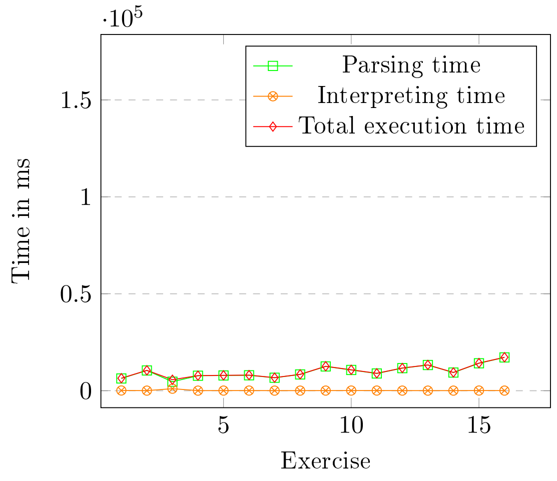
Figure 5.1

### Comparison between parsing and interpreting time

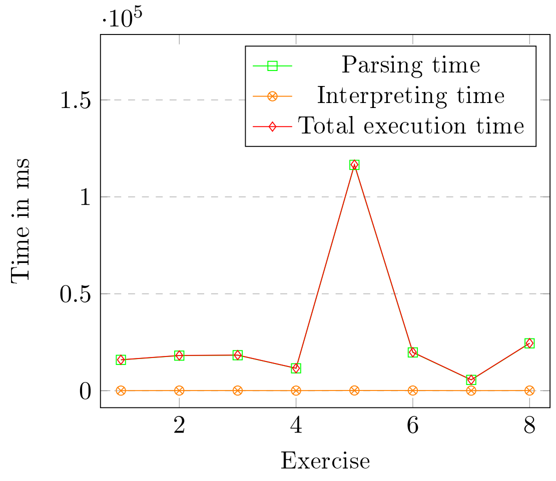
Fig 5.2a and fig 5.2b provide a comparison between the execution time required to parse a submission and the execution time required to interpret a submission for each exercise of both courses. Both figures confirm the fact that the total execution time depends entirely upon the parser slowness. The 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 5.2

## 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 5.1, 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 5.3 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 5.1. 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 5.1: Statistics about correctness

**Submission:**

```
local Fact S in
    fun {Fact N}
      local
        fun {FactList L N Acc}
          if N==1 then L|Acc
          else {FactList L|Acc N-1 N*Acc}
        end
      in
        {Browse {FactList nil N 1}}
      end
    end
  S = {Fact 3}
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 5.3: 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 B. The results are showed in table 5.2 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 B.

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

Table 5.2: 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 5.3 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:**

```
local Fact S in
    fun {Fact N}
      local
        fun {FactList L N Acc}
          if N==1 then L|Acc
          else {FactList L|Acc N-1 N*Acc}
        end
      in
        {Browse {FactList nil N 1}}
      end
    end
  S = {Fact 3}
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:

```
local Fact S in
  fun {Fact N}
    declare
    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
  end
  S = {Fact 3}
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:**

```
    local Fact S in
      fun {Fact N}
        fun {Fact N Acc}
          if N<2 then Acc
          else {Fact N-1 N*Acc} end
        end
      in
        case N of nil then nil
        [] H|T then {Fact H 1}|{Fact T}
        end
      end
      S = {Fact 3}
    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:**

```
local AppendLists R in
  fun {AppendLists L1 L2}
    case L1 of nil then L2
    [] H|T then H|{AppendLists L1 L2}
    end
  end
  R={AppendLists [1] [2]}
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

```
local R Prime in
  fun {Prime N}
    local isPrime 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
      {IsPrime N 1}
    end
  end
  R = {Prime 7}
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 6

# 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 4, 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 5, 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 7

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

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

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

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

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

[9] INGInious Team. Inginious' Documentation. `http://inginious.readthedocs.io/en/latest/index.html`.

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

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

[12] 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.

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

[14] 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.

[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.

# Appendix A

# How to test it

## Jar file

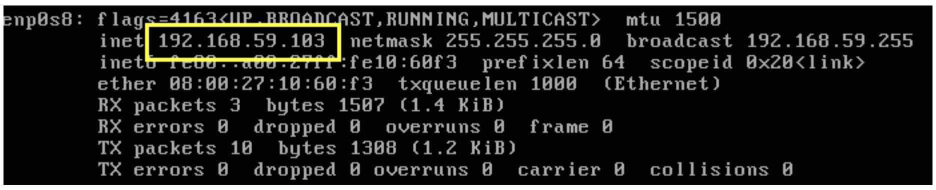
You can find our source code on the following bitbucket git: `https://paquota@bitbucket.org/paquota/correctoz.git`. All the classes you need are regrouped in the *CorrectOz.jar* file, you only have to execute it with the corresponding arguments. This command is the simplest one to run our tool on the Oz code written in the [FileName]. You can obviously use all the others arguments presented previously. A README is provided on the git with all the necessary information. You will also find on the git an example of the current configuration on INGInious and some inputs file containing codes in Oz.

```
java -jar CorrectOz.jar [FileName] -d false
```

## Virtual machine

If you prefer to use our tool via INGInious, we also provide you a VM with the exercises of the first MOOC. You can download the VM (3GB) via this link :
`https://drive.google.com/open?id=0B5SqZj-xejjrWVRmNkRWR04zdXM` All you have to do is to submit a code as a student would do. The procedure to use the virtual machine is the following one:

1. Open it with virtual box
2. Username and password are 'root'
3. Go to the INGInious folder: **cd INGInious**
4. Find your IP address: **ifconfig**



```
enp0s8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.59.103  netmask 255.255.255.0  broadcast 192.168.59.255
        inet6 fe80::a00:27ff:fe10:60f3  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:10:60:f3  txqueuelen 1000  (Ethernet)
        RX packets 3  bytes 1507 (1.4 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 10  bytes 1308 (1.2 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

5. Launch the webapp: **inginious-webapp –host [IpAddress] –port 80**
6. Open your browser and go to the [IpAddress]
7. User and password are 'test'
8. Click on the unique course : [LTEST0000] Test tasks : H2G2 - List of exercises
9. Choose your exercise among the 15 and submit your code.

# Appendix B

# Examples

### 0.0.1 Edx1

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx0 -nVar 1 -nProc 0 -nFun 0 -nCall 1**, corresponding to the command line for the first exercise of the course.

**Example 1** :

```
local X Y Z in
   Y = (6*5)
   Z = 9-7
   X = Y+Z
   {Browse X}
end
```

```
HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 1 variables instead of
 3.
```

**Example 2** :

```
local X in
   Y = (6*5)+(9-7)
   {Browse Y}
end
```

```
ERROR :  The variable is not declared. The variable(s) concerned :[Y].
Error found in " Y = ( 6 * 5 ) + ( 9 - 7 )".
```

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

**Example 3** :

```
local X Useless in

   fun{Useless R}
      R
   end

   X = {Useless 6*5} + {Useless 9-7}
   {Browse X}

end
```

```
HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 1 variables instead of
 2.


HINT :  You seem to have declare too many functions for the exercice.
Maybe some functions are not needed.We expected 0 functions instead of 1.


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
procedure/function. We expected 1 instead of 3.
```

**Example 4** :

```
local x in
   x= (6*5)+(9-7)
   {Browse x}
end
```

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

ERROR :  The variable is not declared. The variable(s) concerned :[x].
Error found in " x = ( 6 * 5 ) + ( 9 - 7 )".
```

### 0.0.2 Edx2

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx1 -d false -nVar 6 -nFun2 -nCall 6**, corresponding to the command line for the second exercise of the course.

**Example 5** :

```
local P Q T X=1 Y=2 Z=3 S in

  fun {P X}
    X*Y+Z
end

  fun {Q Z}
    X*Y+Z
  end

  T = {S 4}

  {Browse T*3}
  {Browse {Q 4} == 6}
  {Browse {P 5} == 2}
end
```

```
ERROR :  The procedure you want to call is not declared.
 Maybe did you mean one of those : {P, Q}.
The variable(s) concerned :[S]. Error found in " { S 4 }".

ERROR :  You have a deadlock, your variable is not initialized and you
only have one thread.
The variable(s) concerned :[T]. Error found in " T * 3".

ERROR :  One of the term is not a number.
The variable(s) concerned :[T (t0) , 3 (3) ]. Error found in " T * 3".


HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 6 variables instead of
 7.
```

We can see here that an error can unfortunately lead to false positive. Since the student calls a function that does not exist, the variable T will not be assigned and therefore the call to browse with the argument "T*3" will also raise errors. We could stop the analyse after the first error found, but most of the times, it is useful to continue to find out others ones.

**Example 6** :

```
local Q T X=1 Y=2 Z=3 in

  fun {P X}
    X*Y+Z
end

  fun {Q Z}
    X*Y+Z
  end

  T = {P 4}

  {Browse T*3}
  {Browse {Q 4} == 6}
  {Browse {P 5} == 2}
end
```

```
ERROR :  The variable is not declared. The variable(s) concerned :[P].
Error found in " fun { P X } X * Y + Z end".
```

Note that, in this case, even if the function P is not declared, the program will execute and give the correct result. When a student forgets to declare a variable, we declare it for him to let the program execute as it should.

**Example 7** :

```
local P Q T X=1 Y=2 Z=3 in

   fun {P X
      X*Y+Z
   end

   fun {Q Z}
      X*Y+Z
   end

   T = {P 4}

   {Browse T*3}
   {Browse {Q 4} == 6}
   {Browse {P 5} == 2}
end
```

```
ERROR :  A " } " is missing in your procedure/function declaration.
Error found in " fun { P X X * Y + Z end".
```

```
ERROR :  Too many argument for the procedure.
The variable(s) concerned :[P]. Error found in " { P 4 }".

ERROR :  Too many argument for the procedure.
The variable(s) concerned :[P]. Error found in " { P 5 }".
```

Once again, the first error leads to two other non correct feedback. But once again, the result is the correct one as we try to correct as much errors as we can while giving feedback to the student.

**Example 8** :

```
local P Q T X=1 Y=2 Z=3 in

   fun {P X}
      X*Y+Z
   end

   fun Q Z}
      X*Y+Z
   end

   T = {P 4}

   {Browse T*3}
   {Browse {Q 4} == 6}
   {Browse {P 5} == 2}
end
```

```
ERROR :  A " { " is missing in your procedure/function declaration.
Error found in " fun Q Z } X * Y + Z end".

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 :[Z]. Error found in " fun Q Z } X * Y + Z end
".

ERROR :  The variable is not a number. The variable(s) concerned :[Z (
OzFunction :
    body = funQZ}X*Y+Zend
    Env = {P=p0, Q=q0, T=t0, X=x0, Y=y0, Z=z0}
)]. Error found in " X * Y + Z".

ERROR :  You have a deadlock, your variable is not initialized and you
only have one thread.
The variable(s) concerned :[T]. Error found in " T * 3".
```

```
ERROR :  One of the term is not a number.
The variable(s) concerned :[T (t0) , 3 (3) ]. Error found in " T * 3".

ERROR :  The procedure you want to call is not declared.
 Maybe did you mean one of these : {P, Z}.The variable(s) concerned :[Q].
 Error found in " { Q 4 }".

ERROR :  You have an error due to one of the variables Error found in " {
 Q 4 } == 6".

ERROR :  You have an error due to one of the variables Error found in " {
 P 5 } == 2".
```

The first error is responsible of all the others. It is a consequence of continuing to look after errors even if we already found one.

### 0.0.3 Edx3

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx3 -d false -nVar 2 -nFun1 -nCall 5**, corresponding to the command line for the third exercise of the course.

**Example 9** :

```
local X SlowAdd in
  fun {SlowAdd Z Y}
         {Delay 1000}
         Z+Y
   end

   X = {SlowAdd 1000 1} + {SlowAdd 1000 1} + {SlowAdd 1000 1}
    {Browse X}
end
```

```
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 "1000" with value "1000" at
position "1" in method "SlowAdd", Argument "1" with value "1" at position
 "2"
in method "SlowAdd"]The variable(s) concerned :[SlowAdd].
Error found in " { SlowAdd 1000 1 }".


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
procedure/function. We expected 5 instead of 7.
```

For the exercise, the fact that the student does not store the value in the variable should be considered as an error, but this is too specific to the exercise to be implemented as such. The unit tests should be in charge to report this error.

**Example 10** :

```
local X SlowAdd in

   fun {SlowAdd Z Y}
         {Delay 1000}
         Z+Y
   end

    X = {SlowAdd}

end
```

```
ERROR :  Not enough argument for the procedure.
The variable(s) concerned :[SlowAdd]. Error found in " { SlowAdd }".

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

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

ERROR :  One of the term is not a number.
The variable(s) concerned :[Z (null) , Y (null) ]. Error found in " Z + Y
".
```

Some false positive due to the first error.

**Example 11** :

```
local X SlowAdd in

   fun {SlowAdd Z Y}
         {Delay 1000}
         Z+Y
   end


    fun {SlowAdd X Y}
         {Delay 1000}
            X + Y
   end

    X={SloawAdd 3*1000 3*1}

    {Browse X}

end
```

```
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 :[SlowAdd]. Error found in " fun { SlowAdd X Y
} { Delay 1000 } X + Y end".

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

ERROR :  The procedure you want to call is not declared.
Maybe did you mean one of those : {SlowAdd}.
The variable(s) concerned :[SloawAdd]. Error found in " { SloawAdd 3 *
1000 3 * 1 }".

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

**Example 12** :

```
local X SlowAdd in

   fun {SlowAdd Z Y}
         {Delay 1000}
         Z+Y
   end


    fun {SlowAdd X Y}
         {Delay 1000}
            X + Y
   end

      fun {SlowAdd X Y}
         {Delay 1000}
       X+Y
   end

   local X in
         fun {SlowAdd X }
            {Delay 3000}
               X = (X+Y)+(X+Y)+(X+Y)

         end

    end
    {Browse {SlowAdd 3003}}

end
```

```
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 :[SlowAdd].
Error found in " fun { SlowAdd X Y } { Delay 1000 } X + Y end".

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 :[SlowAdd].
Error found in " fun { SlowAdd X } { Delay 3000 } X = ( X + Y ) + ( X + Y
 ) + ( X + Y ) end".

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

ERROR :  The variable is not a number.
The variable(s) concerned :[Y (null)]. Error found in " X + Y".
```

```
ERROR :  One of the term is not a number.
The variable(s) concerned :[(X+Y) (null) , (X+Y) (null) ]. Error found in
 " ( X + Y ) + ( X + Y )".

ERROR :  The variable is not a number.
The variable(s) concerned :[(X+Y) (null)]. Error found in " ( X + Y ) + (
 X + Y ) + ( X + Y )".

ERROR :  The variable is not declared.
The variable(s) concerned :[X]. Error found in " X = ( X + Y ) + ( X + Y
) + ( X + Y )".


HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 2 variables instead of
 3.
```

### 0.0.4 Edx4

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx4 -d false -nVar 3 -nFun 2 -nIf 1 -tailRec true**, corresponding to the command line for the exercise number 4 of the course.

**Example 13** :

```
local MainSum R in
   fun {MainSum N}
         local Sum in
               fun {Sum N Acc}
                 if N==0 then 0
            else N*N+{Sum N-1 Acc}
         end
               end
             {Sum N 0}
         end
      end

   R = {MainSum 8}
end
```

```
ERROR : These functions are not tail recursive : [Sum]
```

**Example 14** :

```
local MainSum R in
   fun {MainSum N}
      fun {Sum N Acc}
         if N=0 then Acc
         else {Sum N-1 Acc+N*N}
         end
      end
      in {Sum N 0}
   end

   R = {MainSum 8}

end
```

```
ERROR : Your program contains some errors

ERROR :  The variable is not declared.
The variable(s) concerned :[N]. Error found in " N = 0".

ERROR :  The Condition is not a boolean.
The value is null.  Error found in " if N = 0 then Acc else { Sum N - 1
Acc + N * N } end".
```

**Example 15**

```
local MainSum Sum R in
   fun {MainSum N}
      fun {Sum N Acc}
         if N==0 then {Browse {MainSum n}}
         else Acc=(N)^2 N=N-1
         end
       end
    {Sum N 0}
   end

   R = {MainSum 8}

end
```

```
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.
 The variable(s) concerned :[Acc]. Error found in " Acc = ( N ) ^ 2".
```

```
ERROR :  This operator does not exist in Oz. The variable(s) concerned
:[^]. Error found in " ( N ) ^ 2".

ERROR :  The variable is not declared. The variable(s) concerned :[N].
Error found in " N = N - 1".
```

**Example 16**

```
local MainSum Sum R in
   fun {MainSum N}
      fun {Sum N Acc}
         fun {Sum N Acc}
            if (N==0) then Acc
            else {MainSum N+1 Acc*N} end
         end
       end
    {Sum N 0}
   end

   R = {MainSum 8}
end
```

```
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 :[Sum].
Error found in " fun { Sum N Acc } if ( N == 0 ) then Acc else { MainSum
N + 1 Acc * N } end end".

HINT : You seem to have declared too many functions for the exercise.
Maybe some functions
are not needed. We expected 2 functions instead of 3.
```

**Example 17**

```
local MainSum Sum R in
   fun {MainSum N}
      fun {Sum N Acc}
         if (N==0) Acc
         else Acc= Acc + N ^2
       end
    {Sum N 0}
   end
   R = {MainSum 8}
end
```

```
ERROR :  A " then " statement is missing in the if statement.
[if <condition> then <statement> (else <statement>) end].
Error found in " if ( N == 0 ) Acc else Acc = Acc + N ^ 2".

ERROR :  A " end " statement is missing in the if statement.
 [if <condition> then <statement> (else <statement>) end].
 Error found in " if ( N == 0 ) Acc else Acc = Acc + N ^ 2".


HINT :  An " else " statement is missing in your if structure.
it is better to have a default case.
Error found in " if ( N == 0 ) Acc else Acc = Acc + N ^ 2".


HINT :  The case/if statement does not match any pattern for the
expression.
You should have an else statement.
The variable(s) concerned :[(N==0)( false)].
Error found in " if ( N == 0 ) Acc else Acc = Acc + N ^ 2".
```

The two hints are false positives due to the two first errors.

**Example 18**

```
local MainSum Sum R in
   fun {MainSum N}
      fun {Sum N Acc}
         if (N==0) then Acc
         else Acc= Acc + N ^2 N--
         end
       end
     {Sum N 0}
   end
   R = {MainSum 8}
end
```

```
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.
The variable(s) concerned :[Acc]. Error found in " Acc = Acc + N ^ 2".

ERROR :  This operator does not exist in Oz.
The variable(s) concerned :[^]. Error found in " N ^ 2".

ERROR :  The variable is not a number.
The variable(s) concerned :[N^2 (null)]. Error found in " Acc + N ^ 2".

ERROR :  This operator does not exist in Oz.
The variable(s) concerned :[--]. Error found in " N --".
```

**Example 19**

```
local MainSum Sum R in
   fun {MainSum N}
      fun {Sum N Acc}
         if N == 0 then Acc
         else {SumAcc N-1 Acc+N*N}
       end
    {Sum N 0}
   end
   R = {MainSum 8}
end
```

```
ERROR :  A " end " statement is missing in the if statement.
[if <condition> then <statement> (else <statement>) end].
Error found in " if N == 0 then Acc else { SumAcc N - 1 Acc + N * N }".

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

ERROR :  The procedure you want to call is not declared.
Maybe did you mean one of those : {MainSum, Sum}.
The variable(s) concerned :[SumAcc]. Error found in " { SumAcc N - 1 Acc
+ N * N }".

HINT :  An " else " statement is missing in your if structure.
it is better to have a default case.
 Error found in " if N == 0 then Acc else { SumAcc N - 1 Acc + N * N }".

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

**Example 20**

```
local MainSum Sum R in
   fun {MainSum N}
      fun {Sum N Acc}
         while N >0 then
            Acc=Acc+N*N
            N-1
         end
      end
    {Sum N 0}
   end

   R = {MainSum 8}

end
```

```
ERROR : Your program contains some errors

ERROR :  A " do " statement is missing in the for/while construction.
[for/while <loopDec> do <statement>].
Error found in " while N > 0 then Acc = Acc + N * N N - 1 end".

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.
 The variable(s) concerned :[Acc]. Error found in " Acc = Acc + N * N".

HINT :  You do not use enough if statement for the exercise.
We expect you to use at least 1 if statement(s).
```

### Example 21

```
local MainSum Sum R in
   fun {MainSum N}
      fun {Sum N Acc}
         if N==0 then Acc
         else{Sum N-1 AccN*N}
         end
      end
    {Sum N 0}
   end

   R = {MainSum 8}

end
```

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

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (8) ]. Error found in " AccN * N".

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (7) ]. Error found in " AccN * N".

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (6) ]. Error found in " AccN * N".

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (5) ]. Error found in " AccN * N".

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (4) ]. Error found in " AccN * N".
```

```
ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (3) ]. Error found in " AccN * N".

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (2) ]. Error found in " AccN * N".

ERROR :  One of the term is not a number. The variable(s) concerned :[
AccN (null) , N (1) ]. Error found in " AccN * N".
```

**Example 22**

```
local MainSum Sum R in
   fun {MainSum N}
      fun {Sum N Acc}
         N*N
         end
    {Sum N 0}
   end

   R = {MainSum 8}

end
```

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

HINT :  You do not use enough if statement for the exercise. We expect
you to use at least 1 if statement(s).
```

We can see here the utility of simple check such as the number of if structure used. In this case, the student is warned that he should use at least one if structure for the exercise. The exercise is therefore more complicated that he thought and he is redirected in the good direction with the two hints.

**Example 23**

```
local MainSum Sum R in
   fun {MainSum N}
      fun {Sum N Acc}
         if N==0 then Acc
         else {Acc=N^2} {N=N-1} end

         end
    {Sum N 0}
   end

   R = {MainSum 8}

end
```

```
ERROR :  This operator does not exist in Oz.
The variable(s) concerned :[^]. Error found in " N ^ 2".

ERROR :  The variable is not declared.
The variable(s) concerned :[Acc]. Error found in " Acc = N ^ 2".

ERROR :  The procedure you want to call is not declared.
Maybe did you mean one of those : {MainSum, Sum}.
The variable(s) concerned :[Acc=N^2]. Error found in " { Acc = N ^ 2 }".

ERROR :  The variable is not declared.
The variable(s) concerned :[N]. Error found in " N = N - 1".

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

In this case, the student has not understood the notation for the call of procedure. Even if the feedback is not as precise as we would, it can lead the student to understand that the notation he used is for procedure call.

### 0.0.5 Edx5

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx5 -d false -nVar 4 -nFun 2 -nCall 7 -nIf 1 -tailRec true**, corresponding to the command line for the exercise number 5 of the course.

**Example 24** :

```
local R Prime in
   fun {Prime N}
         fun {PrimeAcc N Acc}
            if N==1 then false
               else if Acc==1 then true
                  else if (N mod Acc)==0 then false
                     else {PrimeAcc N Acc}
                     end
                  end
               end
            end
         in {PrimeAcc N N-1}
   end
   R = {Prime 7}
end
```

```
ERROR :  Your program runs infinitely, it can be from one of
these parts of the program :
[ fun { PrimeAcc N Acc } if N == 1 then false else if Acc == 1 then
true else if ( N mod Acc ) == 0 then false else { PrimeAcc N Acc }
end end end end]
You call Argument "N" with value "7" at position "1" in method "PrimeAcc"
 187 times.
Maybe you forgot to update it.
You call Argument "Acc" with value "6" at position "2" in method "
PrimeAcc" 186 times.
 Maybe you forgot to update it.

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 "N" with value "7" at position "1" in
method "PrimeAcc",
Argument "Acc" with value "6" at position "2" in method "PrimeAcc"].
The variable(s) concerned :[PrimeAcc]. Error found in " { PrimeAcc N Acc
}".
```

## Example 25 :

```
local R Prime in
   fun {Prime N}
      local isPrime 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
      {IsPrime N 1}
      end
   end
R = {Prime 7}
end
```

```
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 " 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 }".
```

**Example 26** :

```
local R Prime in
   fun {Prime N}
      fun {While N S}
          if N = 1 then false
          elseif N = S then true
          elseif (N mod S) = 0 then false
          elseif 1 = 1 then {While N (S+1)}
          end
   end
   {While N 1}
end
R = {Prime 7}
end
```

```
ERROR : Your program contains some errors

ERROR :  The variable is not declared. The variable(s) concerned :[While
].
Error found in " fun { While N S } if N = 1 then false elseif N = S then
true
elseif ( N mod S ) = 0 then false elseif 1 = 1 then { While N ( S + 1 ) }
 end end".

ERROR :  The variable is not declared. The variable(s) concerned :[N].
Error found in " N = 1".

ERROR :  The Condition is not a boolean. The value is null.
Error found in " if N = 1 then false elseif N = S then true elseif
( N mod S ) = 0 then false elseif 1 = 1 then { While N ( S + 1 ) } end".

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 false elseif N = S
then true elseif ( N mod S ) = 0 then false elseif 1 = 1
 then { While N ( S + 1 ) } end".
```

### 0.0.6 Edx6

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx6 -d false -nVar 3 -nFun 2 -nIf 1 -tailRec true**, corresponding to the command line for the exercise number 6 of the course.

**Example 27** :

```
local Fib R in
    fun {Fib N}
        local FibAux in
            fun {FibAux N Acc1 Acc2}
                if N == 0 then 0
               else if N == 1 then 1
                  else Acc1+{FibAux N Acc2 Acc1+Acc2}
                  end
               end
            end
            {FibAux N 0 1}
      end
    end
    R = {Fib 9}
end
```

```
ERROR : These functions are not tail recursive : [FibAux]

ERROR :  Your program runs infinitely, it can be from one of these parts
of the program :
[ fun { FibAux N Acc1 Acc2 } if N == 0 then 0 else if N == 1 then 1
else Acc1 + { FibAux N Acc2 Acc1 + Acc2 } end end end]
You call Argument "Acc2" with value "1" at position "2" in
method "FibAux" 2 times. Maybe you forgot to update it.
You call Argument "N" with value "9" at position "1" in
method "FibAux" 223 times. Maybe you forgot to update it.
```

**Example 28** :

```
local Fib R in
    fun {Fib N}
        local FibAux in
            fun {FibAux N Acc1 Acc2}
               if N == 0 then 0
               else  if N==1 then 1
               else  {FibAux N-1 0 1}+ {FibAux N-2 0 1}
               end
         end
          {FibAux N 0 1}
      end
      end
   R = {Fib 9}
end
```

```
ERROR : These functions are not tail recursive : [FibAux]

ERROR :  A " end " statement is missing in the if statement. [if <
condition> then <statement> (else <statement>) end]. Error found in " if
N == 0 then 0 else if N == 1 then 1 else { FibAux N - 1 0 1 } + { FibAux
N - 2 0 1 } end".

===== HINTS =====

HINT :  An " else " statement is missing in your if structure. it is
better to have a default case. Error found in " if N == 0 then 0 else if
N == 1 then 1 else { FibAux N - 1 0 1 } + { FibAux N - 2 0 1 } end".


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 "N-1" with value "1" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 1 0 1 }".


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 "N-2" with value "0" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 2 0 1 }".


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 "N-1" with value "2" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 1 0 1 }".


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 "N-2" with value "1" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 2 0 1 }".


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 "N-1" with value "3" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 1 0 1 }".


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 "N-2" with value "2" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 2 0 1 }".


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 "N-1" with value "4" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 1 0 1 }".


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 "N-2" with value "3" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"]The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 2 0 1 }".


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 "N-1" with value "5" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"] The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 1 0 1 }".


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 "N-2" with value "4" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"] The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 2 0 1 }".


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 "N-1" with value "6" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"] The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 1 0 1 }".


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 "N-2" with value "5" at position "1" in method
"FibAux", Argument "0" with value "0" at position "2" in method "FibAux",
 Argument "1" with value "1" at position "3" in method "FibAux"] The
variable(s) concerned :[FibAux]. Error found in " { FibAux N - 2 0 1 }".


HINT : The variable Acc2 is not used. It is probably useless.


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

The problem here is that the student does not use the accumulator, resulting in calling the function several times with the same arguments leading to this big feedback. Since he calls the functions twice at every call with the same accumulator that he does not update, CorrectOz think it could store the result in a variable. Even if this feedback is not the correct improvement for the mistake, the student is warned that he calls several times the same functions with the same arguments and that the two accumulators are not used. This feedback could make him find the right answer.

**Example 29** :

```
local Fib R in
    fun {Fib N}
        local FibAux in
            fun {FibAux N Acc1 Acc2}
            if N==0 then Acc1
            else
             {FibAux Acc2 Acc2+Acc1}
            end
         end
          {FibAux N 0 1}
      end
      end
   R = {Fib 9}
   {Browse R}
end
```

```
ERROR :  Not enough argument for the procedure.
The variable(s) concerned :[FibAux].
Error found in " { FibAux Acc2 Acc2 + Acc1 }".

ERROR :  The variable is not declared.
The variable(s) concerned :[Acc2]. Error found in " Acc2".

ERROR :  One of the term is not a number.
The variable(s) concerned :[Acc2 (null) , Acc1 (1) ].
Error found in " Acc2 + Acc1".

ERROR :  You have a deadlock, your variable is not initialized and you
only have one thread. The variable(s) concerned :[N].
Error found in " N == 0".

ERROR :  One of the term is not a number.
The variable(s) concerned :[Acc2 (null) , Acc1 (-43) ].
Error found in " Acc2 + Acc1".

ERROR :  Your program runs infinitely, it can be from
one of these parts of the program :
[ fun { FibAux N Acc1 Acc2 } if N == 0 then Acc1 else { FibAux Acc2 Acc2
+ Acc1 } end end]
You call Argument "Acc2" with value "null" at position "1" in method "
FibAux" 368 times. Maybe you forgot to update it.
You call Argument "Acc2+Acc1" with value "-43" at position "2" in method
"FibAux" 368 times. Maybe you forgot to update it.

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 "Acc2" with value "null" at position "1" in
```

```
 method
"FibAux", Argument "Acc2+Acc1" with value "-43" at position "2" in method
 "FibAux"].
The variable(s) concerned :[FibAux]. Error found in " { FibAux Acc2 Acc2
+ Acc1 }".
```

We have some false positive in the feedback of this exercise. Since the student forgot one argument when calling the function, the program thinks that the missing argument is not declared and has therefore the null value. This leads to a wrong feedback about the declaration of the variable and the deadlock. However, since the real error is explained (the number of argument is not the right one), the student can recover easily.

**Example 30** :

```
local Fib R in
    fun {Fib N}
        local FibAux in
            fun {FibAux N Acc1 Acc2}
            if N == 0 then 0 else 1 end
         end
          {FibAux N 0 1}
      end
      end
   R = {Fib 9}
   {Browse R}
end
```

```
HINT : The variable Acc2 is not used. It is probably useless.

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

In this case, even if the syntax is correct and without unit test, the feedback shows to the student that his program does not behave as it should do since it does not use the two accumulators. If we couple it with some unit tests, CorrectOz would be really powerful.

**Example 31** :

```
local Fib R in
    fun {Fib N}
        local FibAux in
            fun {FibAux N Acc1 Acc2}
            if N==0 then Acc2
            else {FibAux N-1 Acc1 Acc1+Acc2} + {FibAux N-2 Acc1+Acc2 Acc2
            }
            end
         end
```

```
            {FibAux N 0 1}
        end
        end
    R = {Fib 9}
    {Browse R}
end
```

```
ERROR: These functions are not tail recursive: [FibAux]

ERROR: Your program runs infinitely, it can be from one of those parts of
 the program: [ fun { FibAux N Acc1 Acc2 } if N == 0 then Acc2 else {
FibAux N - 1 Acc1 Acc1 + Acc2 } + { FibAux N - 2 Acc1 + Acc2 Acc2 } end
end]
You call Argument "Acc1+Acc2" with value "1" at position "3" in method "
FibAux" 9 times. Maybe you forgot to update it.

You call Argument "Acc1" with value "0" at position "2" in method "FibAux
" 9 times. Maybe you forgot to update it.

You call Argument "Acc1" with value "1" at position "2" in method "FibAux
" 320 times. Maybe you forgot to update it.
```

**Example 32** :

```
local Fib R in
    fun {Fib N}
        local FibAux in
            fun {FibAux N Acc1 Acc2}
            if (N==1) then 1
            elseif (N==0) then 0
            else {FibAux N-1 } + {FibAux N-2 } end
        end
            {FibAux N 0 1}
        end
        end
    R = {Fib 9}
end
```

```
ERROR: These functions are not tail recursive: [FibAux]

ERROR: Not enough argument for the procedure. The variable(s) concerned
:[FibAux]. Error found in " { FibAux N - 1 }".

ERROR: Not enough argument for the procedure. The variable(s) concerned
:[FibAux]. Error found in " { FibAux N - 2 }".
```

```
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 "N-1" with value "1" at position "1" in
method "FibAux"]

The variable(s) concerned:[FibAux]. Error found in " { FibAux N - 1 }".

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 "N-2" with value "0" at position "1" in
method "FibAux"]

The variable(s) concerned:[FibAux]. Error found in " { FibAux N - 2 }".

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 "N-1" with value "2" at position "1" in
method "FibAux"]
The variable(s) concerned:[FibAux]. Error found in " { FibAux N - 1 }".

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 "N-2" with value "1" at position "1" in
method "FibAux"]
The variable(s) concerned:[FibAux]. Error found in " { FibAux N - 2 }".

HINT: The variable Acc2 is not used. It is probably useless.

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

**Example 34** :

```
local Fib R in
    fun {Fib N}
        local FibAux in
            fun {FibAux N Acc1 Acc2}
            if N==0 then Acc1+Acc2
            else {FibAux N Acc1+Acc2 Acc1}
            end
          {FibAux N 0 1}
      end
      end
   R = {Fib 9}
```

```

end
```

```
 ERROR: A " end " statement is missing in the if statement. [if <
 condition> then <statement> (else <statement>) end]. Error found in " if
  N == 0 then Acc1 + Acc2 else { FibAux N Acc1 + Acc2 Acc1 }".

ERROR: Your program runs infinitely, it can be from one of these parts of
 the program:
[ fun { FibAux N Acc1 Acc2 } if N == 0 then Acc1 + Acc2 else { FibAux N
Acc1 + Acc2 Acc1 } end]

You call Argument "Acc1+Acc2" with value "1" at position "2" in method "
FibAux" 2 times. Maybe you forgot to update it.

You call Argument "Acc1" with value "1" at position "3" in method "FibAux
" 2 times. Maybe you forgot to update it.

You call Argument "N" with value "9" at position "1" in method "FibAux"
363 times. Maybe you forgot to update it.

HINT: An " else " statement is missing in your if structure. it is better
 to have a default case. Error found in " if N == 0 then Acc1 + Acc2 else
 { FibAux N Acc1 + Acc2 Acc1 }".
```

**Example 35** :

```
local Fib R in
    fun {Fib N}
        local FibAux in
            fun {FibAux N Acc1 Acc2}
            if N == 0 then Acc1
            else {FibIter N-1 Acc2 Acc1+Acc2} end
            end
          {FibAux N 0 1}
      end
      end
   R = {Fib 9}
   {Browse R}
end
```

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

ERROR :  The procedure you want to call is not declared. Maybe did you
mean one of these : {FibAux, Fib}.
```

```
The variable(s) concerned :[FibIter]. Error found in " { FibIter N - 1
Acc2 Acc1 + Acc2 }".

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

### 0.0.7 Edx7

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx7 -d false -nVar 2 -nFun1 -nCase 1 -nilCase true -list true -extProc [Append]**, corresponding to the command line for the exercise number 7 of the course.

**Example 36** :

```
local AppendLists R in

   fun {AppendLists L1 L2}
      case L1 of nil then L2
       H|T then H|{AppendLists T L2}
      end
   end

R={AppendLists [1] [2]}
end
```

```
HINT :  A " [] " or an " else " statement is missing in the case
structure. [case <expression> of <pattern> then <statement> { [] <pattern
> then <statement> } || { else <statement> } end]. Error found in " case
L1 of nil then L2 H | T then H | { AppendLists T L2 } end".

HINT :  The case/if statement does not match any pattern for the
expression. You should have an else statement. The variable(s) concerned
:[L1 ([ 1 ] )]. Error found in " case L1 of nil then L2 H | T then H | {
AppendLists T L2 } end".


HINT :  Some functions are not tail recursive.[AppendLists]
```

**Example 37** :

```
local AppendLists R in

   fun {AppendLists L1 L2}
      case L1 of nil then L2
      [] H|T then H|{AppendLists L1 L2}
      end
   end

R={AppendLists [1] [2]}
end
```

```
ERROR :  Your program runs infinitely, it can be from one of these 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" 308 times. Maybe you forgot to update it.
You call Argument "L1" with value "[ 1 ]" at position "1" in method "
AppendLists" 308 times. Maybe you forgot to update it.


===== HINTS =====

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.


HINT :  Some functions are not tail recursive.[AppendLists]
```

**Example 38** :

```
local AppendLists R in

   fun {AppendLists L1 L2}
      {Append L1 L2}
   end

R={AppendLists [1] [2]}
end
```

```
ERROR :  You cannot use this predifined function.
The variable(s) concerned :[Append]. Error found in " { Append L1 L2 }".
```

```
HINT :  You do not use enough case statements for the exercise.
We expect you to use at least 1 case statement(s).

HINT :  You do not have a " case <var> of nil " statement.
You should at least have one to succeed this exercise.
```

**Example 39** :

```
local AppendLists R in

   fun {AppendLists L1 L2}
         L1|L2
   end

R={AppendLists [1] [2]}
end
```

```
HINT :  You do not use enough case statements for the exercise.
We expect you to use at least 1 case statement(s).


HINT :  You do not have a " case <var> of nil " statement.
You should at least have one to succeed this exercise.
```

Our feedback, even if they seem silly at the first sight can help the student to understand that he is not in the right direction.

**Example 40** :

```
local AppendLists R in

   fun {AppendLists L1 L2}
      if {L1==L2==nil} then nil
      elseif {L1 == nil} then L2
      elseif {L2 == nil} then L1
      else L1 | L2| nil
      end

   end

R={AppendLists [1] [2]}
end
```

```
ERROR : Your program contains some errors
ERROR :  One of the term is not a number nor a boolean. Error found in "
L1 == L2 == nil".

ERROR :  The procedure you want to call is not declared. Maybe did you
mean one of these : {AppendLists}.
The variable(s) concerned :[L1==L2==nil]. Error found in " { L1 == L2 ==
nil }".

ERROR :  The Condition is not a boolean. The value is null.  Error found
in " if { L1 == L2 == nil } then nil elseif { L1 == nil } then L2 elseif
{ L2 == nil } then L1 else L1 | L2 | nil end".

HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 1 case statement(s).

HINT :  You do not have a " case <var> of nil " statement. You should at
least have one to succeed this exercise.
```

**Example 41** :

```
local AppendLists R in

   fun {AppendLists L1 L2}
      local Addelement List Newlist in
          fun {Addelement List Newlist}
              if List == nil then Newlist|L2
               else {Addelement List.2 Newlist|List.1}
               end
            end
            {Addelement  L1 L2}
      end
   end

R={AppendLists [1] [2]}
end
```

```
ERROR :  Your program runs infinitely, it can be from one of these parts
of the program :
[ fun { Addelement List Newlist } if List == nil then Newlist | L2 else {
 Addelement List . 2 Newlist | List . 1 } end end,

The list you build does not contain a nil : Newlist|List.1]

HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 2 variables instead of
 5.

HINT :  You do not use enough case statements for the exercise.
```

```
We expect you to use at least 1 case statement(s).

HINT : You do not have a " case <var> of nil " statement.
You should at least have one to succeed this exercise.
```

**Example 42** :

```
local AppendLists R in

   fun {AppendLists L1 L2}
      if L1==nil then L2
      else L1|L2.1 {AppendLists L1 L2.2}
      end
   end

R={AppendLists [123] [3 4 5]}
end
```

```
ERROR : Your program contains some errors

ERROR :  You are trying to access the field of an element which is not a
record. The variable(s) concerned :[L2 (nil]. Error found in " L1 | L2 .
1".

ERROR :  You are trying to access the field of an element which is not a
record. The variable(s) concerned :[L2 (nil)]. Error found in " {
AppendLists L1 L2 . 2 }".

HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 1 case statement(s).

HINT :  You do not have a " case <var> of nil " statement. You should at
least have one to succeed this exercise.
```

**Example 43** :

```
local AppendLists R in

fun{AppendLists L1 L2}
L1 | L2 | nil;
end

R = {AppendLists [1 2 3] [4 5 6]}
end
```

```
ERROR :  The end of a statement is not marked by a ';' in Oz. Error found
 in " L1 | L2 | nil ;".

HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 1 case statement(s).

HINT :  You do not have a " case <var> of nil " statement. You should at
least have one to succeed this exercise.
```

### 0.0.8 Edx8

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx8 -d false -nVar 3 -nFun2 -nIf 1 -list true -tailRec true**, corresponding to the command line for the exercise number 8 of the course.

**Example 44** :

```
local Fact S in
   fun {Fact N}
      fun {Fact N Acc}
            if N<2 then Acc
            else {Fact N-1 N*Acc} end
       end
      in
         case N of nil then nil
               [] H|T then {Fact H 1}|{Fact T}
          end
   S = {Fact 12}
end
```

```
ERROR : These functions are not tail recursive : [Fact]

ERROR :  A " end " statement is missing in the case structure. [case <
expression> of <pattern> then <statement> end]. Error found in " case N
of nil then nil [] H | T then { Fact H 1 } | { Fact T }".

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 :  The case/if statement does not match any pattern for the
expression. You should have an else statement. The variable(s) concerned
:[N (12 )]. Error found in " case N of nil then nil [] H | T then { Fact
H 1 } | { Fact T }".
```

**Example 45** :

```

local Fact S in
   fun {Fact N}
      fun {Fact2 N A}
         if N<2 then Acc
          else {Fact2 N-1 N*A} end
      end
      in
         case L of nil then nil
                  [] H|T then {Fact2 H 1}|{Fact T}
         end
   end
   S = {Fact 3}
end
```

```
ERROR : These functions are not tail recursive : [Fact]

ERROR : Your program contains some errors

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

HINT : The variable Acc is not used. It is probably useless.

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

**Example 46** :

```

local Fact S in
   fun {Fact N}
      local fct in
         fun {fct A B Acc}
             if A == N then Acc
             else {fct A+1 A*B {Append A*B Acc}}
             end
         end
         {fct 1 1 nil}
      end
   S = {Fact 3}
end
```

```
ERROR :  A " end " statement is missing in the local declaration. [local
X1.....XN in <statement> end]. Error found in " local fct".
```

```
ERROR :  A " in " statement is missing in the local declaration. [local
X1.....XN in <statement> end]. Error found in " local fct".

ERROR :  A variable should always start with an upper case. Error found
in " fun { fct A B Acc } if A == N then Acc else { fct A + 1 A * B {
Append A * B Acc } } end end".

ERROR :  The variable is not declared. The variable(s) concerned :[fct].
Error found in " fun { fct A B Acc } if A == N then Acc else { fct A + 1
A * B { Append A * B Acc } } end end".

ERROR :  One of the two arguments given to the function Append is not a
list. The variable(s) concerned :[A*B (1 ), Acc (nil) ]. Error found in "
 { Append A * B Acc }".

ERROR :  One of the two arguments given to the function Append is not a
list. The variable(s) concerned :[A*B (2 ), Acc (arg6) ]. Error found in
" { Append A * B Acc }".

ERROR :  You have a deadlock, your variable is not initialized and you
only have one thread. The variable(s) concerned :[Acc]. Error found in "
Acc".
```

**Example 47** :

```

local Fact S in
   fun {Fact N}
      local
          fun {Factorial N A}
              if N == 1 then A
               else {Factorial N - 1 A * N}
              end
          end

      fun {FactorialAux A L}
        if A == 1 then {Append L 1}
        else {Factorial A-1 {Append L Factorial{A 1}}}
        end
    end
in
    {List.reverse {Fact N nil}}
end
      end
   S = {Fact 8}
end
```

```
ERROR :  The variable is not declared. The variable(s) concerned :[
Factorial]. Error found in " fun { Factorial N A } if N == 1 then A else
{ Factorial N - 1 A * N } end end".

ERROR :  The variable is not declared. The variable(s) concerned :[
FactorialAux]. Error found in " fun { FactorialAux A L } if A == 1 then {
 Append L 1 } else { Factorial A - 1 { Append L Factorial { A 1 } } } end
 end".

ERROR :  Too many arguments for the procedure. The variable(s) concerned
:[Fact]. Error found in " { Fact N nil }".

ERROR :  Your program runs infinitely, it can be from one of these parts
of the program :
[ fun { Factorial N A } if N == 1 then A else { Factorial N - 1 A * N }
end end,  fun { Fact N } local fun { Factorial N A } if N == 1 then A
else { Factorial N - 1 A * N } end end fun { FactorialAux A L } if A == 1
 then { Append L 1 } else { Factorial A - 1 { Append L Factorial { A 1 }
} } end end in { List . reverse { Fact N nil } } end end]
You call Argument "N" with value "8" at position "1" in method "Fact" 338
 times. Maybe you forgot to update it.

HINT : The variable List is not used. It is probably useless.

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

**Example 48** :

```

local Fact S in
   fun {Fact N}
      local FactN L J in
           fun {FactN I A}
             if I==0 then A
             else {FactN I-1 I*A}
             end
           end
         if N==0 then L
           else
             J={FactN N 1}
           end
         end
      end
   S = {Fact 8}
end
```

```
HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 3 variables instead of
```

```
 5.

HINT :  You have to use some list structure in this exercise.
```

### 0.0.9 Edx9

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx9 -d false -nVar 3 -nFun2 -nIf 1 -list true -tailRec true**, corresponding to the command line for the exercise number 9 of the course.

**Example 50** :

```

local Prefix FindString R in

declare
fun {Prefix L1 L2}
   case L1#L2
   of nil#Ys then true
   [] Xs#nil then false
   [] nil#nil then true
   [] (X|Xs)#(Y|Ys) then
      if X==Y then {Prefix Xs Ys}
      else false end
   end
end


declare
fun {FindString L1 L2}
   if {Prefix L1 L2} then true
   else
      case L2
      of nil then false
      [] (X|Xs) then {FindString L1 Xs}
      end
   end
end
R = {FindString [1 4] [2 1 4 6]}
end
```

```
ERROR : Your program contains some errors

ERROR :  You should use a local instead of declare.  Error found in "
declare fun { Prefix L1 L2 } case L1 # L2 of nil # Ys then true [] Xs #
nil then false [] nil # nil then true [] ( X | Xs ) # ( Y | Ys ) then if
```

```
X == Y then { Prefix Xs Ys } else false end end end declare fun {
FindString L1 L2 } if { Prefix L1 L2 } then true else case L2 of nil then
 false [] ( X | Xs ) then { FindString L1 Xs } end end end".

ERROR :  You should use a local instead of declare.  Error found in "
declare fun { FindString L1 L2 } if { Prefix L1 L2 } then true else case
L2 of nil then false [] ( X | Xs ) then { FindString L1 Xs } end end end
".

ERROR :  The Condition is not a boolean. The value is null.  Error found
in " if { Prefix L1 L2 } then true else case L2 of nil then false [] ( X
| Xs ) then { FindString L1 Xs } end end".

HINT :  The case/if statement does not match any pattern for the
expression. You should have an else statement. The variable(s) concerned
:[L1#L2 (nil#[ 6 ] )]. Error found in " case L1 # L2 of nil # Ys then
true [] Xs # nil then false [] nil # nil then true [] ( X | Xs ) # ( Y |
Ys ) then if X == Y then { Prefix Xs Ys } else false end end".
```

**Example 51** :

```

local Prefix FindString R in

fun{Prefix L1 L2}
    case L1#L2 of nil#L2 then true
    [](H1|T1)#(H2|T2) then
        if H1==H2 then {Prefix T1 T2}
        else false
        end
    end
end
fun{FindString L1 L2}
    if {Prefix L1 L2} then true
    else
        case L1#L2 of nil#nil then true
        [] L1#(H2|T2) then {FindString L1 T2}
        else false
        end
    end
end
R = {FindString [1 4] [2 1 4 6]}
end
```

```
ERROR : Your program contains some errors

ERROR :  The Condition is not a boolean. The value is null.  Error found
in " if { Prefix L1 L2 } then true else case L1 # L2 of nil # nil then
true [] L1 # ( H2 | T2 ) then { FindString L1 T2 } else false end end".
```

```
HINT :  The case/if statement does not match any pattern for the
expression. You should have an else statement. The variable(s) concerned
:[L1#L2 (nil#[ 6 ] )]. Error found in " case L1 # L2 of nil # L2 then
true [] ( H1 | T1 ) # ( H2 | T2 ) then if H1 == H2 then { Prefix T1 T2 }
else false end end".
```

### 0.0.10 Edx10

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx10 -d false -nFun 2 -nCase 1 -nilCase true -list true -tailRec true -extProc [Flatten]** , corresponding to the command line for the exercise number 10 of the course.

**Example 52** :

```
local FlattenList S in
   fun {FlattenList L}
      local FlattenListBis in
         fun {FlattenListBis L Acc}
               case L of M then M|Acc
               [] nil then Acc
               [] H|T then {FlattenListBis T {Append
{FlattenListBis H nil} Acc}}
               end
         end
            {FlattenListBis L nil}
      end
   end
   S= {FlattenList [[1 2] [5 6]]}
end
```

```
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 is a dead code. The case statement is 'case L of M'. Error found
 in " case L of M then M | Acc [] nil then Acc [] H | T then {
FlattenListBis T { Append { FlattenListBis H nil } Acc } } end".

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

ERROR :  You have a deadlock, your variable is not initialized and you
only have one thread. The variable(s) concerned :[L]. Error found in "
case L of M then M | Acc [] nil then Acc [] H | T then { FlattenListBis T
 { Append { FlattenListBis H nil } Acc } } end".
```

**Example 53** :

```
local FlattenList S in
   fun {FlattenList L}
      case L of nil then L
         [] H|T then {Append H {FlattenList T}}
      end
   end
   S= {FlattenList [ 1 [a2 3] [4] [[5 6]]]}
end
```

```
ERROR: One of the two arguments given to the function Append is not a
list.
The variable(s) concerned:[H (1 ), {FlattenListT} ([ 2 3 4 [ 5 6 ] ]) ].
Error found in " { Append H { FlattenList T } }".
```

**Example 54** :

```
local FlattenList S in
   fun {FlattenList L}
      case L of H|T then H|{FlattenList T}
            [] H|nil then L
      end
   end
   S= {FlattenList [ 1 [a2 3] [4] [[5 6]]]}
end
```

```
ERROR : These functions are not tail recursive : [FlattenList]

ERROR : Your program contains some errors

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

**Example 55** :

```
local FlattenList S in
   fun {FlattenList L}
      case L of nil then L
               []H|T then {Append {Flatten H}{Flatten
T}}
            else H|{Flatten T}
      end
   end
   S= {FlattenList [ 1 [a2 3] [4] [[5 6]]]}
end
```

```
ERROR: You cannot use this predifined function. The variable(s) concerned
 :[Flatten]. Error found in " { Flatten H }".
```

### 0.0.11 Edx11

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx11 -d false -nFun 2 -nCase 1 -list true**, corresponding to the command line for the exercise number 11 of the course.

**Example 56** :

```
local FunnyFunc R in
   fun {FunnyFunc FunL L}
       case L of H|T then {FunL.1 H}|{FunnyFunc FunL.2 T}
       else nil
       end
   end

   R = {FunnyFunc [fun {$ X} X+1 end fun {$ X} X*2 end] [2 2 2]}
end
```

```
ERROR : Your program contains some errors

ERROR :  You are trying to access the field of an element which is not a
record.
The variable(s) concerned :[FunL (nil) ]. Error found in " { FunL . 1 H
}".

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

```
ERROR :  You are trying to access the field of an element which is not a
record.
 The variable(s) concerned :[FunL (nil) ]. Error found in " { FunnyFunc
 FunL . 2 T }".


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 "H" with value "2" at position "1" in
method "$"]
The variable(s) concerned :[FunL.1]. Error found in " { FunL . 1 H }".


HINT :  You seem to have declare to many functions for the exercice.
Maybe some functions are not needed.We expected 2 functions instead of 3.


HINT :  Some functions are not tail recursive.[FunnyFunc]
```

**Example 57** :

```
local FunnyFunc Test in
   fun {FunnyFunc FunL L}
       case L of H|T then {FunL.1 H}|{FunnyFunc FunL.2 T}
       else nil
       end
   end

    proc {Test FunL L SolL}
    {Browse {FunnyFunc FunL L} == SolL}
   end

{Test [fun($ X) X*X end fun($ X) X*X*X end fun($ X) X*3 end ] [4 2 3] [16
 8 9]}
end
```

```
ERROR: A " { " is missing in your procedure/function declaration. Error
found
in " fun ( $ X ) X * X end".

ERROR: A " } " is missing in your procedure/function declaration. Error
found
in " fun ( $ X ) X * X end".

ERROR: A " { " is missing in your procedure/function declaration. Error
found
in " fun ( $ X ) X * X * X end".

ERROR: A " } " is missing in your procedure/function declaration. Error
found
in " fun ( $ X ) X * X * X end".

ERROR: A " { " is missing in your procedure/function declaration. Error
found
in " fun ( $ X ) X * 3 end".

ERROR: A " } " is missing in your procedure/function declaration. Error
found
in " fun ( $ X ) X * 3 end".
```

### 0.0.12 Edx12

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx12 -d false -nFun3 -nIf 1 -nCase 1 -list true -bottom true -tailRec true**, corresponding to the command line for the exercise number 12 of the course.

**Example 58** :

```
local F Build D C in
   fun {Build D C}
      fun {$ X}
         fun {FindX X D C}
             case D of H|T then
                if H==X then C.1
                else {FindX X D C.2}
                end
            else bottom
            end
         end
      in {FindX X D C}
      end
   end
   D = [1 2 3]
   C = [4 5 6]
   F = {Build D C}
   {Browse {F 1}}
   {Browse {F 2}}
   {Browse {F 3}}
   {Browse {F 4}}
end
```

```
ERROR :  You are trying to access the field of an element which is not a
record. The variable(s) concerned :[C (nil) ]. Error found in " { FindX X
 D C . 2 }".

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

### 0.0.13 Edx13

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx13 -d false -nFun2 -nCase 1 -nilCase true -list true -tailRec true**, corresponding to the command line for the exercise number 13 of the course.

**Example 59** :

```
local Transform R in
   fun{Transform L}
      local TransformValues in
         fun {TransformValues R F V}
               case V of nil then R
               [] H|T then
                   R.(F.1) = {Transform H}
                   {TransformValues R F V}
               end
         end
         case L of [G F V] then {TransformValues {Record.make G
F} F V}
         else L
         end
      end
   end
R = {Transform [record [3 5 6] [b c d]]}
end
```

```
ERROR :  Your program runs infinitely, it can be from one of these parts
of the program :
[ fun { TransformValues R F V } case V of nil then R [] H | T then R . (
F . 1 ) = { Transform H } { TransformValues R F V } end end]
You call Argument "F" with value "[ 3 5 6 ]" at position "2" in method "
TransformValues" 367 times. Maybe you forgot to update it.
You call Argument "V" with value "[ b c d ]" at position "3" in method "
TransformValues" 367 times. Maybe you forgot to update it.
You call Argument "R" with value "record({3=b, 5=_, 6=_})" at position
"1" in method "TransformValues" 366 times. Maybe you forgot to update it.
You call Argument "H" with value "b" at position "1" in method "Transform
" 367 times. Maybe you forgot to update it.
```

```
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 "H" with value "b" at position "1" in method "
Transform"]The variable(s) concerned :[Transform]. Error found in " {
Transform H }".
```

```
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 "R" with value "record({3=b, 5=_, 6=_})" at
position "1" in method "TransformValues", Argument "F" with value "[ 3 5
6 ]" at position "2" in method "TransformValues", Argument "V" with value
 "[ b c d ]" at position "3" in method "TransformValues"]The variable(s)
concerned :[TransformValues]. Error found in " { TransformValues R F V
}".
```

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

**Example 60** :

```
local Transform R in
   fun{Transform L}
      local R Recordness Add Acc
      in
          fun {Add R Lol Acc}
               case Lol of H|T then
                  if Acc<{Width R} then R.Acc=Lol.Acc
                  else R
                     end
               end
            end

         fun {Recordness Lis}
            R = {Record.make L.1 L.2.1}
            case L.2.2 of H|T then {Recordness T}
             else {Add R T 1}
               end
         end
         {Recordness L}
   end
   end
R = {Transform [record [3 5 6] [b c d]]}
end
```

```
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. The variable(s)
concerned :[R]. Error found in " R = { Record . make L . 1 L . 2 . 1 }".

ERROR :  Your program runs infinitely, it can be from one of these parts
of the program : [ fun { Recordness Lis } R = { Record . make L . 1 L . 2
 . 1 } case L . 2 . 2 of H | T then { Recordness T } else { Add R T 1 }
end end]

HINT :  A " [] " or an " else " statement is missing in the case
structure. [case <expression> of <pattern> then <statement> { [] <pattern
> then <statement> } || { else <statement> } end]. Error found in " case
Lol of H | T then if Acc < { Width R } then R . Acc = Lol . Acc else R
end end".

HINT : The variable Lis is not used. It is probably useless.

HINT :  You do not have a " case <var> of nil " statement. You should at
least have one to succeed this exercise.
```

### 0.0.14 Edx14

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx14 -d false -nVar 5 -nFun3 -nIf 1 -nCase 3 -nilCase true -leafCase true -list true -tailRec true**, corresponding to the command line for the exercise number 14 of the course.

**Example 61** :

```
local Infix R Tree in
   fun {Infix Tree}
      case Tree of nil then nil
      [] btree(T left:L right:R) then {Append {Infix L} T|{Infix R}}
      end
   end

   Tree = btree(4 left:btree(2 left:btree(1 left:leaf right:leaf)
                                  right:btree(3 left:leaf right:leaf))
                  right:btree(5 left:leaf right:leaf))

   R = {Infix Tree}
end
```

```
HINT :  The case/if statement does not match any pattern for the
expression.
You should have an else statement. The variable(s) concerned :[Tree (leaf
 )].
Error found in " case Tree of nil then nil [] btree ( T left : L right :
R ) then { Append { Infix L } T | { Infix R } } end".
```

```
HINT :  You do not use enough if statement for the exercise. We expect
you to use at least 1 if statement(s).
```

```
HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 3 case statement(s).
```

```
HINT :  You do not have a \" case <var> of leaf \" statement. You should
at least have one to succeed this exercise.
```

more informations

**Example 62** :

```
local Infix R Tree in
   fun {Infix Tree}

declare
fun{Infix T Acc}
   if(T.left\=leaf) then
      {Infix T.left Acc}
      Acc|T.1
   elseif(T.right \= leaf) then
      {Infix T.right Acc}
   else
      Acc
   end
end

declare
T = btree(4 left:btree(2 left:btree(1 left:leaf right:leaf)
                                   right:btree(3 left:leaf right:leaf))
     right:btree(5 left:leaf right:leaf))
{Browse {Infix T nil}}  end


   Tree = btree(1:42 left:btree(1:12 left:btree(1:9 left:leaf right:leaf)
right:btree(1:21 left:leaf right:btree(1:24 left:leaf right:btree(1:31
left:leaf right:leaf)))) right:leaf)

   R = {Infix Tree}
end
```

```
RROR :  You should use a local instead of declare.
Error found in " declare fun { Infix T Acc } if ( T . left \= leaf ) then
 { Infix T . left Acc } Acc | T . 1 elseif ( T . right \= leaf ) then {
Infix T . right Acc } else Acc end end declare T = btree ( 4 left : btree
 ( 2 left : btree ( 1 left : leaf right : leaf ) right : btree ( 3 left :
```

```
 leaf right : leaf ) ) right : btree ( 5 left : leaf right : leaf ) )".

ERROR :  You should use a local instead of declare.  Error found in "
declare T = btree ( 4 left : btree ( 2 left : btree ( 1 left : leaf right
 : leaf ) right : btree ( 3 left : leaf right : leaf ) ) right : btree (
5 left : leaf right : leaf ) )".

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 :[Infix]. Error found in " fun
{ Infix T Acc } if ( T . left \= leaf ) then { Infix T . left Acc } Acc |
 T . 1 elseif ( T . right \= leaf ) then { Infix T . right Acc } else Acc
 end end".

ERROR :  The variable is not declared. The variable(s) concerned :[T].
Error found in " T = btree ( 4 left : btree ( 2 left : btree ( 1 left :
leaf right : leaf ) right : btree ( 3 left : leaf right : leaf ) ) right
: btree ( 5 left : leaf right : leaf ) )".


HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 3 case statement(s).


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


HINT :  You do not have a \" case <var> of leaf \" statement. You should
at least have one to succeed this exercise.
```

### 0.0.15 Edx15

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx15 -d false -nFun3 -nIf 1 -nCase 3 -nilCase true -leafCase true -list true -tailRec true**, corresponding to the command line for the exercise number 15 of the course.

**Example 63** :

```
local FromListToTree R T FromTreeToList S in


fun{FromListToTree L}
    local AddE AddAllL in
        fun{AddE T E}
            case T
            of leaf then btree(E left:leaf right:leaf)
            [] btree(K left:T1 right:T2) andthen K==E then
                T
            [] btree(K left:T1 right:T2) andthen K>E then
                btree(K left:{AddE T1 E} right:T2)
            [] btree(K left:T1 right:T2) andthen K<E then
                btree(K left:T1 right:{AddE T2 E})
            end
        end
        fun{AddAllL L T}
            if L==nil then T
            else {AddAllL L.2 {AddE T L.1}} end
        end
        {AddAllL L leaf}
    end
end

fun{FromTreeToList T}
    local RemoveSmallest DoAllT in
        fun{RemoveSmallest T}
            case T
            of leaf then none
            [] btree(K left:T1 right:T2) then
                case {RemoveSmallest T1}
                of none then double(T2 K)
                [] double(Ts Xs) then
                    double(btree(K left:Ts right:T2) Xs)
                end
            end
        end
        fun{DoAllT T}
            case {RemoveSmallest T}
            of none then nil
            [] double(Ts Xs)
                Xs|{DoAllT Ts}
            end
        end
        {DoAllT T}
    end
end

    R = {FromListToTree [42 21 12 34 9]}
    T = btree(1:42 left:btree(1:9 left:leaf right:leaf) right:leaf)
    S = {FromTreeToList T}


end
```

```
ERROR : These functions are not tail recursive : [DoAllT]

ERROR :  A " then " statement is missing in the case structure. [case <
expression> of <pattern> then <statement> end]. Error found in " case {
RemoveSmallest T } of none then nil [] double ( Ts Xs ) Xs | { DoAllT Ts
} end".
```

```
HINT :  The case/if statement does not match any pattern for the
expression. You should have an else statement. The variable(s) concerned
:[{RemoveSmallestT} (double({1=btree({1=42, left=leaf, right=leaf}),
2=9}) )]. Error found in " case { RemoveSmallest T } of none then nil []
double ( Ts Xs ) Xs | { DoAllT Ts } end".
```

```
HINT :  You seem to have declare too many variables for the exercice.
Probably some variable are not needed. We expected 5 variables instead of
 9.
```

```
HINT :  You do not have a " case <var> of nil " statement. You should at
least have one to succeed this exercise.
```

### 0.0.16 Edx2_1

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_1 -d false -nFun3 -nIf 1 -nCase 2 -leafCase true -tailRec true**, corresponding to the command line for the exercise number 1 of the second course.

**Example 65** :

```
local Reverse L  Y Z R T S in
   fun{Reverse L}
      local X in
          {NewCell nil X}
         for E in L do
            X:=E|@X
            end
            X
      end
   end

L ={Reverse [1 2 3]}
{Browse L.1}
end
```

```
ERROR :  You are trying to access the field of an element which is a cell
.
You probably want to access to the value of the cell which is a record.
Try with the " @ " operator. The variable(s) concerned :[L (OzCell : [ 3
2 1 ] )].
Error found in " { Browse L . 1 }".
```

**Example 66** :

```
local Reverse Z in
   fun{Reverse L}
      local X in
          {NewCell nil X}
         for E in L do
            X:=E|X
            end
            @X
      end
   end

Z= {Reverse [1 2 3]}
{Browse Z}

end
```

```
ERROR :  Your program runs infinitely, it can be from one of these parts
of the program :
[The list you build does not contain a nil : E|X]


HINT :  You do not use enough if statement for the exercise. We expect
you to use at least 1 if statement(s).


HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 2 case statement(s).


HINT :  You do not have a \" case <var> of leaf \" statement. You should
at least have one to succeed this exercise.
```

**Example 67** :

```
local Reverse Y in
   Y = {NewCell 128}
   Y:= Y+12

end
```

```
ERROR :  One of the term of you operation is a cell. Use the " @ "
operator to access its content.
The variable(s) concerned :[Y]. Error found in " Y + 12".
```

```
HINT :  You do not use enough if statement for the exercise. We expect
you to use at least 1 if statement(s).


HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 2 case statement(s).


HINT :  You do not have a \" case <var> of leaf \" statement. You should
at least have one to succeed this exercise.
```

**Example 68**

```
local Reverse L in
   fun{Reverse L}
      for E H T in L do
          if L==nil then nil
          elseif L==H|T then E={Reverse T}|H
          end
      end
   end
L ={Reverse [1 2 3]}
end
```

```
ERROR : These functions are not tail recursive : [Reverse]

ERROR :  You can only use one variable in the for loop. The correct
syntax is "for <var> in <var> do <statement> end". Error found in " E H T
 in L".

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

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

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


HINT :  An " else " statement is missing in your if structure.
it is better to have a default case. Error found in " if L == nil then
nil elseif L == H | T then E = { Reverse T } | H end".


HINT :  You do not use enough case statements for the exercise.
We expect you to use at least 2 case statement(s).


HINT :  You do not have a \" case <var> of leaf \" statement.
```

```
You should at least have one to succeed this exercise.
```

## Example 69

```
local Reverse L in
   fun{Reverse L}

L = {NewCell nil}
for E in L do
     if (E==nil) then @L
     L=@L|E
end
   end

L ={Reverse [1 2 3]}
end
```

```
ERROR : Your program contains some errors

ERROR :  A " end " statement is missing in the if statement.
[if <condition> then <statement> (else <statement>) end].
Error found in " if ( E == nil ) then @ L".

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.
The variable(s) concerned :[L]. Error found in " L = { NewCell nil }".

ERROR :  You want to iterate over the value of a cell. Are you sure you
did not forget to use the " @ " operator ? Error found in " E in L".

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

HINT :  An " else " statement is missing in your if structure. it is
better to have a default case. Error found in " if ( E == nil ) then @ L
".


HINT :  You do not use enough case statements for the exercise. We expect
 you to use at least 2 case statement(s).


HINT :  You do not have a \" case <var> of leaf \" statement. You should
at least have one to succeed this exercise.
```

### 0.0.17 Edx2_2

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_2 -d false -nFun 1 -tailRec true -cell true**, corresponding to the command line for the exercise number 2 of the second course.

**Example 70** :

```
local Shuffle R in
    fun {Shuffle L}
      local
            Size = {Length L}
            Array = {NewArray 1 Size 0}
            Output = {NewCell nil}
        in
            for I in 1..Size do
            Array.I:= {Nth L I}
         end
            local
                Max = {NewCell Size}
            in
                for I in 1..Size do
                    local RandomNumber = {OS.rand} mod @Max + 1 in
                    Output:= Array.RandomNumber | @Output
                    Array.RandomNumber:= Array.@Max
                    Max:= @Max - 1
                    end
                end
            end
            @Output
      end
   end

R:= {Shuffle [1 2 3 4 5]}
{Browse R}
end
```

```
ERROR :  The variable is not a cell/attribute. You cannot use the " := "
operator.
The variable(s) concerned :[R]. Error found in " R := { Shuffle [ 1 2 3 4
 5 ] }".
```

**Example 71** :

```
local Shuffle R in
    fun {Shuffle L}
      local
            Size = {Length L}
            Array = {NewArray 1 Size 0}
            Output = {NewCell nil}
        in
            for I in 1..Size do
            Array.I:= {Nth L I}
         end
            local
                Max = {NewCell Size}
            in
                for I in 1..Size
                    local RandomNumber = {OS.rand} mod @Max + 1 in
                    Output:= Array.RandomNumber | @Output
                    Array.RandomNumber:= Array.@Max
                    Max:= @Max - 1
                    end
                end
            end
            @Output
      end
   end

R:= {Shuffle 1}
{Browse R}
end
```

```
ERROR : Your program contains some errors

ERROR :  A " do " statement is missing in the for/while construction. [
for/while <loopDec> do <statement>]. Error found in " for I in 1 .. Size
local RandomNumber = { OS . rand } mod @ Max + 1 in Output := Array .
RandomNumber | @ Output Array . RandomNumber := Array . @ Max Max := @
Max - 1 end end".

ERROR :  The variable is not a cell/attribute. You cannot use the " := "
operator. The variable(s) concerned :[R]. Error found in " R := { Shuffle
 1 }".

ERROR :  The argument of Length is not a list nor a string. The variable(
s) concerned :[L (1) ]. Error found in " { Length L }".

HINT :  You have to use some cell structure in this exercise.
```

### 0.0.18 Edx2_3

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_3 -d false -nFun1 -nFun 4 -nilCase true -list true -cell true**, corresponding to the command line for the exercise number 3 of the second course.

**Example 72** :

```
local NewStack IsEmpty Push Pop Eval R in
   fun {NewStack}
      {NewCell nil}
   end

      fun {IsEmpty S}
      case S of nil then true
      else false
      end
      end

   proc {Push S X}
      S:= X|@S
   end

   fun {Pop S}
      if {IsEmpty S} then empty
      else
         case @S of H|T then
            S:= T
            H
         else
            empty
            end
         end
      end

   fun {Eval L}
      local S={NewStack} in
         for E in L do
            case E
            of '+' then {Push S {Pop S}+{Pop S}}
            [] '-' then {Push S ~{Pop S}+{Pop S}}
            [] '*' then {Push S {Pop S}*{Pop S}}
            [] '/' then
               local Tmp = {Pop S} in
                  {Push S {Pop S} div Tmp}
                  end
            else {Push S E}
            end
         end
         {Pop S}
      end
   end
      R = {Eval [1 8 '+' 4 '*' 10 '/'] }
   {Browse R}
end
```

```
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 ? Error found in " case S of nil then
true else false end".
```

### 0.0.19 Edx2_4

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_4 -d false -nClass 1 -list true**, corresponding to the command line for the exercise number 4 of the second course.

**Example 73** :

```
local Collection C C2 X in
   class Collection
      attr list

      meth init
         list:= nil
      end

      meth put(X)
         list:=X|@list
      end

      meth isEmpty($)
         @list==nil
      end

   meth get($)
      case @list of H|T then list:=T
                          H
      [] nil then notfound
      end
   end

   meth browse
      {Browse @list}
   end


```

```
     meth union(C)
       if {C isEmpty($)}==false then
          {self put({C get($)})}
          {self union(C)}
       end
    end

    end

    C = {New Collection init}
    {C put 3}



end
```

```
===== ERRORS FOUND =====
ERROR: You have to put the argument of the method between parenthesis: {C
put(3)}. Error found in " { C put 3 }".

ERROR: Not the right number of arguments for the method. Error found in "
 { C
put 3 }".
```

**Example 74**

```
local Collection C C2 X in
   class Collection
      attr list

      meth init
         list:= nil
      end

      meth put(X)
         list:=X|@list
      end

      meth isEmpty($)
         @list==nil
      end

   meth get($)
      case @list of H|T then list:=T
                           H
      [] nil then notfound
      end
   end

   meth browse
```

```
      {Browse @list}
   end


    meth union(C)
      if {C isEmpty($)}==false then
         {self put({C get($)})}
         {self union(C)}
      end
   end

   end

   C = {New Collection init}
   {C put}

end
```

```
===== ERRORS FOUND =====
ERROR: Not the right number of arguments for the method. Error found in "
 { C
put }".
```

### 0.0.20 Edx2_5

Here follow some examples of code and feedback given when executing CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_5 -d false -nClass 1 -list true**, corresponding to the command line for the exercise number 5 of the second course.

**Example 75** :

```
local Constant Variable VarX VarY Addition Substraction Multiplication
Division
in
      class Constant
         attr c
         meth init(X) c:=X end
         meth evaluate($) @c end
      end

      class Variable
         attr v
         meth init(X) c:=X end
         meth set(X) c:=X end
         meth evaluate($) @c end
      end

      class Addition
         attr a b
         meth init(X Y) a:=X b:=Y end
         meth evaluate($) {@a evaluate($)}+{@b evaluate($)} end
      end

      class Substraction
         attr a b
         meth init(X Y) a:=X b:=Y end
         meth evaluate($) {@a evaluate($)}-{@b evaluate($)} end
      end

      class Multiplication
         attr a b
         meth init(X Y) a:=X b:=Y end
         meth evaluate($) {@a evaluate($)}*{@b evaluate($)} end
      end

      class Division
         attr a b
         meth init(X Y) a:=X b:=Y end
         meth evaluate($) {@a evaluate($)} div {@b evaluate($)}
end
      end

      VarX = {New Variable init(5)}
        VarY = {New Variable init(6)}
        local
            Result
            C = {New Constant init(6)}
            Expr1 = {New Addition init(VarX VarY)}
            Expr2 = {New Division init(Expr1 C)}
        in
            {VarX set(6)}
            {VarY set(2)}
            {Expr2 evaluate(Result)}
            {Browse Result}
        end


end
```

```
===== ERRORS FOUND =====
ERROR: The cell/attribute c is not declared. Error found in " c:= X".

ERROR: One of the term is not a number Error found in " { @ a evaluate (
$ ) }
+ { @ b evaluate ( $ ) }".

ERROR: One of the two terms is not a number. Error found in " { @ a
evaluate (
$ ) } div { @ b evaluate ( $ ) }".

ERROR: the variable Result is not declared. Error found in " { Browse
Result
}".
```

**Example 76** :

```
local Constant Variable VarX VarY Addition Substraction Multiplication
Division
in
      class Constant
         attr c
         meth init(X) c:=X end
         meth evaluate($) @c end
      end

      class Variable
         attr v
         meth init(X) v:=X end
         meth set(X) v:=X end
         meth evaluate($) @v end
      end

      class Addition
         attr a b
         meth init(X Y) a:=X b:=Y end
         meth evaluate($) {a evaluate($)}+{@b evaluate($)} end
      end

      class Substraction
         attr a b
         meth init(X Y) a:=X b:=Y end
         meth evaluate($) {@a evaluate($)}-{@b evaluate($)} end
      end

      class Multiplication
         attr a b
         meth init(X Y) a:=X b:=Y end
         meth evaluate($) {@a evaluate($)}*{@b evaluate($)} end
      end

      class Division
         attr a b
         meth init(X Y) a:=X b:=Y end
         meth evaluate($) {@a evaluate($)} div {@b evaluate($)}
end
      end

      VarX = {New Variable init(0)}
        VarY = {New Variable init(0)}
        local
            Result
            C = {New Constant init(6)}
            Expr1 = {New Addition init(VarX VarY)}
            Expr2 = {New Division init(Expr1 C)}
        in
            {VarX set(1)}
            {VarY set(4)}
            {Expr2 evaluate(Result)}
            {Browse Result}
        end


end
```

```
===== ERRORS FOUND =====
ERROR: You are trying to call a method on a cell. did not you forget to
access
the value of the cell thanks to the "@" operator ? Error found in " { a
evaluate
( $ ) }".
```

### 0.0.21 Edx2_7

Here follow some examples of code and feedback given when executing the CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_7 -d false -nFun 1 -nCall 4 -list true -cell true**, corresponding to the command line for the exercise number 7 of the second course.

**Example 77** :

```
 local Reverse in
   fun {Reverse S}

      local I={NewCell S} in
         local Temp={NewCell nil} in
            for C in I do
                Temp:=C|@Temp
            end
            @Temp
         end
      end

   end

   {Browse {Reverse [1 2 3 4 5]}}
end
```

```
===== ERRORS FOUND =====
ERROR: You want to iterate over the value of a cell. Are you sure you did
 not
forget to use the "@" operator ? Error found in " C in I".
```

Even if the student made a mistake by iterating on the cell instead of the value of the cell, we do it for him and the program continues to execute normally, searching for other mistakes.

### 0.0.22 Edx2_8

Here follow some examples of code and feedback given when executing the CorrectOz with the following command line: **java -jar CorrectOz.jar edx2_8 -d false -nFun 1 -nCall 4 -list true -cell true**, corresponding to the command line for the exercise number 8 of the second course.

```
local Stack Queue MyQ MyS Size in

   class Stack
      attr s
      meth init s:=nil end
      meth size($) {Length @s} end
      meth isEmpty($) @s==nil end
      meth browse {Browse @s} end
      meth top($)
          case @s
          of nil then raise emptyStackException() end
          else @s.1
          end
      end
      meth push(X) s:=X|@s end
      meth pop($)
          case @s
          of nil then raise emptyStackException() end
          [] H|T then s:=T H
          end
      end
   end



   class Queue
      attr q
      meth init q:=nil end
      meth size($) {Length @q} end
      meth isEmpty($) @q==nil end
      meth front($)
          case @q
          of nil then raise emptyQueueException() end
          else {Reverse @q}.1
```

```
            end
        end
        meth enqueue(X) q:=X|@q end
        meth browse {Browse @q} end
        meth dequeue($)
            case @q of nil then raise emptyQueueException() end
            else
                local Temp={Reverse @q} in
                    q:={Reverse Temp.2}
                    Temp.1
                end
            end
        end
    end

    MyQ = {New Queue init}
    {MyQ size(Size)}
    {Browse Size}
    {Browse {MyQ isEmpty($)}}
    {MyQ enqueue(3)}
    {Browse {MyQ size($)}}
    {Browse {MyQ isEmpty($)}}
    {MyQ enqueue(5)}
    {Browse {MyQ size($)}}
    {Browse {MyQ isEmpty($)}}
    {Browse {MyQ front($)}}
    {Browse {MyQ dequeue($)}}
    {MyQ browse}
    {Browse {MyQ dequeue($)}}
    {MyQ browse}
    {Browse {MyQ front($)}}
    {Browse {MyQ dequeue($)}}

    MyS = {New Stack init}
    {Browse {MyS size($)}}
    {Browse {MyS isEmpty($)}}
    {MyS push(5)}
    {MyS push(6)}
    {MyS push(7)}
    {Browse {MyS pop($)}}
    {Browse {MyS size($)}}
    {Browse {MyS isEmpty($)}}
    {Browse {MyS top($)}}

end
```

```
===== ERRORS FOUND =====
ERROR: Your code raise an "emptyQueueException()" in the code:" case @ q
of
nil then raise emptyQueueException ( ) end else local Temp = { Reverse @
q } in
```

```
q:= { Reverse Temp . 2 } Temp . 1 end end"
```

## Limitations

Here are different examples where CorrectOz shows its limitations. Some errors cannot unfortunately be detected as we would like.

**Limitation - 1**

```
local MainSum Sum R in
   fun {MainSum N}
      fun {Sum N Acc}
         if N == 0 then Acc
         else {Sum N-1 Acc + (N*2)}
         end
      end
    {Sum N 0}
   end

   R = {MainSum 8}

end
```

Here an example of a submission where the code is correct but the result is not the expected one. Since the error is specific to the exercise, we cannot provide a relevant feedback, we should have implemented a tool for each exercise to detect those errors, but it is not the purpose of CorrectOz.

**Limitation - 2**

```
fun {Infix Tree}
local F A in
    fun{F A}
        if(Tree.left==leaf)
        then nil
        else
            {Append Tree.left A}
            {Indix Tree.left A}
        end
        if(Tree.right==leaf)
        then nil
        else
            {Append A Tree.right}
            {Indix Tree.right A}
            if(Tree.right==leaf)
                if(Tree.right==leaf)
                then nil
                end
            end
        end
    end
    {F Tree.1}
end
```

When the code is too messy and full of syntax errors, CorrectOz does not succeed to parse and to identify the origin of the errors.

**Limitation - 3**

```
local X SlowAdd in

   fun {SlowAdd X Y}
         {Delay 1000}
         X+Y
   end

    X = SlowAdd{3000 + 3}
    Browse(X)

end
```

Some errors of understanding are not identified by CorrectOz, the students are really creative and it is almost impossible to cover all the possible errors.

**Limitation - 4**

```
local X SlowAdd in

   fun {SlowAdd X Y}
         {Delay 1000}
         X+Y
   end

    local X in
    X = {SlowAdd 1000 1}
    {Browse X} + {Browse X} + {Browse X}
end

end
```

The tool does not understand what the student is trying to do and does not succeed to parse the code.

**Limitation - 5**

```
fun {Prime}
   local Prime1 in
       fun{Prime1 N Acc}
             if N==1 then false
            elseif Acc==N then false
            elseif length(N div Acc)==length(N) then false
            else {Prime1 N (Acc+1)}
              end
          end
      end
end
```

The auxiliary function is only called inside itself and is therefore useless. CorrectOz does not take into account this kind of error.

**Limitation - 6**

```
local X in
    X == 2 ;
    fun {Prime1 N X }
        if X == N then true ;
        elseif N mod X ==0 then false
            else {Prime1 N X+1 }
        end
    end
end
```

In some case, when the student (used to java) ends the line with a ";", CorrectOz does not succeed to parse the code. Note that it depends where the semi colon is put.

**Limitation - 7**

```
fun {PrimePart Acc}
    if N % 2 == 0 then false
    else if Acc > N / 2 then true
        else PrimePart Acc + 2
        end
    end
end
```

Here the student used the modulo operator "%" of other language than Oz. Unfortunately, "%" is used in OZ for comments, therefore CorrectOz does not parse this code correctly and does not report the error.

**Limitation - 8**

```
locale Pr in
   fun {Pr N A}
       if A == 1 then true
       else
         if N mod A == 0 then false
         else
            {Pr N A-1}
         end
       end
    end
    {Pr N N div 2}
```

```
end
```

We tried to include some typo faults in our interpreter, but the students always find a way to surprise us. In this case, the locale is not reported by CorrectOz.

**Limitation - 9**

```
 local Fib R in
    fun {Fib N}
        local FibAux in
            fun {FibAux N Acc1 Acc2}
             if N==0 then Acc1+Acc2
            else FibAux N-1 Acc1+Acc2 Acc1 end
               end
            end
            {FibAux N 0 1}
      end
    end
    R = {Fib 9}
   {Browse R}
end
```

CorrectOz does not parse the code correctly when the student forgets to use the bracket to perform a procedure call.

**Limitation - 10**

```
 local Fib R in
    fun {Fib N}
        local FibAux in
            fun {FibAux N Acc1 Acc2}
         if N == 1 then Acc1 end
         if N == 1 then Acc2
         else  {FibAux N-1 Acc1 Acc1+Acc2}
         end
            end
            {FibAux N 0 1}
      end
    end
    R = {Fib 9}
   {Browse R}
end
```

In this case the student use the same condition for two consecutive if statement. CorrectOz does not detect that error.

**Limitation - 11**

```

 local Fib R in
    fun {Fib N}
```

```
        local FibAux in
            fun {FibAux N Acc1 Acc2}
            if N==0 then Acc2
            else
                {FibAux N-1 Acc2 Acc2*Acc1}
            end
         end
            {FibAux N 0 1}
      end
    end
    R = {Fib 9}
   {Browse R}
end
```

The student multiplies his accumulator by 0 at the first iteration, the accumulator will therefore always be equal to zero. it is difficult for us to detect those errors since we cannot know if it is the desired behaviour of the program or not.

# Appendix C

# Statistics

| **Exercise** | **# Students** | **# Submissions** | **Submissions/Students** |
|---|---|---|---|
| FromImplicitToExplicit | 515 | 2,067 | 4 |
| Shuffle | 438 | 1,585 | 3.61 |
| Calculator | 440 | 2,095 | 4.76 |
| Collections | 446 | 2,510 | 5.62 |
| Expressions | 420 | 1,843 | 4.38 |
| Palindrome | 371 | 1,428 | 3.84 |
| FromJavaToOz | 399 | 2,094 | 5.24 |
| Exceptions | 397 | 1,654 | 4.16 |
| ProducerConsumerFilter | 419 | 2,259 | 5.39 |
| TrackingInfo | 376 | 1,824 | 4.85 |
| ForCollectA | 433 | 1,755 | 4.05 |
| IDontWantPrimeNumbers | 399 | 1,547 | 3.87 |
| NFullAdder | 359 | 1,939 | 5.4 |
| InformationExchanges | 178 | 628 | 3.52 |
| TreesAsObjects | 367 | 2,082 | 5.67 |
| WeNeedExceptions | 367 | 1,696 | 4.62 |
| SharedStream | 393 | 700 | 1.78 |
| ObjectsConsumer | 384 | 853 | 2.2 |

Table C.1: Statistical table concerning Louv1.2x exercises

| **Exercise** | **# Students** | **# Submissions** | **Average submissions/student** |
|---|---|---|---|
| HelloWorld | 1,292 | 2,443 | 1.89 |
| BrowseX | 1,270 | 2,152 | 1.69 |
| Scope | 1,198 | 2,821 | 2.34 |
| CalledOnlyOnce | 1,162 | 4,179 | 3.59 |
| Sum | 1,002 | 4,895 | 5.88 |
| Mirror | 949 | 3,721 | 3.92 |
| Prime | 883 | 5,525 | 5.8 |
| Fib | 831 | 3,055 | 3.67 |
| Append | 823 | 4,688 | 5.69 |
| Fact | 743 | 3,770 | 5.07 |
| FindString | 668 | 3,393 | 5.07 |
| Flatten | 344 | 1,273 | 3.7 |
| FunAsInput | 624 | 2,350 | 3.76 |
| BuildMyFunction | 587 | 2,558 | 4.35 |
| BuildMyRecord | 554 | 2,532 | 4.57 |
| Infix | 562 | 2,344 | 4.17 |
| SortWithTree | 493 | 2,448 | 4.96 |
| IsBalanced | 513 | 2,630 | 5.12 |
| Complexity | 463 | 2,033 | 4.39 |
| Midterm | 518 | 1,574 | 3.03 |

Table C.2: Statistical table concerning Louv1.1x exercises

# Appendix D

# Louv1.1x - Paradigms of Computer Programmings: Fundamentals

## 1 Introduction

### 1.1 Why and how should we learn about paradigms ?

A PROGRAM is basically a set of instructions solving a specific problem. These instructions can be written in many different LANGUAGES with the help of many different PARADIGMS.

**Programming paradigm** An approach to programming a computer is based on a coherent set of principles or a mathematical theory. A programming paradigm corresponds to a "*way of thinking*" in order to solve some kinds of problems.

The key idea to visualize here is the 3-level hierarchy programmers are facing everyday:

- Programmers can learn hundreds of LANGUAGES to write diverse programmes.
- Even if many languages **seem** completely different in terms of syntax, they often solve the same kind of problems. We can say these languages realize the same PARADIGM. That is why there are many fewer paradigms than languages.
- Finally, as said above, each paradigm is based on a set of CONCEPTS. In fact, there exists only a few concepts that can be combined to create diverse paradigms. So, we have once again many fewer concepts than paradigms.

As you can see, mastering many languages can be reduced to the understanding of a few paradigms - and even a fewer number of concepts behind these paradigms. Therefore, this course is based on one RESEARCH LANGUAGE called OZ which can express many paradigms at once. To combine many different paradigms, Oz is using a KERNEL LANGUAGE to gather the core concepts behind these many different paradigms. Indeed, each paradigm has its KERNEL LANGUAGE as its simple core language.

**Kernel language** A simple language containing the essential concepts - the primitive elements - of a paradigm.

Louv1.1x covers a first fundamental paradigm to understand a lot more afterwards (cfr. Louv1.2x). This paradigm is called FUNCTIONAL PROGRAMMING. The objective of the course is to build and understand its kernel language concept by concept.

### 1.2 Practical Organization

The course is divided into 6 lessons of one week. Each lesson is explained in one or more videos focusing on one subject at a time. Students can also purchase the course textbook called "*Concepts, Techniques, and Models of Computer Paradigms*, PETER VAN ROY, SEIF HARIDI, MIT Press".

Weekly exercises have to be performed by students on the INGINIOUS paltform in order to improve their practical understanding of the theoretical lessons. These weekly exercises are accounting for 50% of the final grade and students have an infinite number of tries per exercise. This means they can submit unlimited number of submissions on the INGINIOUS platform.

At the end of the course, a final exam is organized to evaluate the remaining 50% of the final grade. During this final exam, the students are limited to 2 submissions only on INGINIOUS. Once the course successfully completed, edX can deliver 2 kinds of certificate to prove the student's achievements:

**Honor Code Certificate** for free registrations.

**Verified Certificate** for registrations performed with a donation.

## 2 Basic programming concepts

### 2.1 A first paradigm

There are 2 fundamental ways of programming which are respectively called IMPERATIVE PROGRAMMING and DECLARATIVE PROGRAMMING.

**Declarative programming** *What* result a programmer wants in terms of properties and the computer figures out *how* to get there. Declarative programming is a good vision to follow in the future of computing. It is for example used in machine learning where computer learns how to solve some problems. Speaking about properties involve speaking in terms of mathematical functions and relations. Therefore, a declarative program that works for one day will work forever. Indeed, mathematical functions do not change. Finally, declarative programming is stateless.

**Imperative programming** It is about telling commands to be computed directly by the computer with a stateful programming in mind.

As stated previously, the objective of the course is to build a fundamental KERNEL LANGUAGE such that its core concepts are equivalent for a lot of others paradigms (and so, for many many more languages). This kernel language realizes a paradigm called FUNCTIONAL PROGRAMMING.

**Functional programming** A form of DECLARATIVE PROGRAMMING which is one of the simplest paradigm and the foundation of many many other ones.

## 2.2 Single and local assignment

Let's first introduce how to DECLARE, ASSIGN and print a variable - we say BROWSE in Oz. Four concepts are essential to understand how these ideas are handled inside the memory:

**Identifier** A character string starting with an uppercase letter which is bound to a variable in memory. This variable is accessible by refering to its given identifier until we redefine it.

**Environment** The binding, the link between a given identifier and a variable in memory. Each point in a program's execution has its own environment. We can consider this environment as a function that takes an identifier as input and returns a variable in memory as output.

**Variable in memory** A programmer does not see this variable but it exists *behind the curtain*. He can access a variable inside the current environment by using its identifier.

**Single assignment** A variable in memory can only be bound to one value.

Single assignment is a fundamental concept of the functional paradigm. Even if it could seem like a big handicap, it is actually *not* the case. In fact, single assignment allows programmers to reduce the risk of breaking a correct program. Indeed, it avoids programmers to mistakenly change the value of a variable.

Even if variables can only be bound to one value, identifiers can refer to multiple values at different parts of the execution. Indeed, at two points in time, the same identifier could refer to different variables in memory. We say they belong to different environments or in other words, they are bound LOCALLY to different variables in memory, each with its own value. The DECLARE instruction allows programmers to instantiate different environments. Indeed, each DECLARE corresponds to one environment at a time. Another way of instantiating an environment is to use a LOCAL instruction in which the environment starts right after the **in** keyword and ends just before the **end** keyword.

A significant concept underlying environments is the SCOPE of an identifier occurence. It is the fifth concept of functional programming.

**Scope** the part of the program code in which that same identifier refers to the same variable in memory.

If you ever wrote any code in Oz, you may have noticed that variables are not staticly typed. Indeed, Oz supports DYNAMIC TYPING which means that the variable type is only defined at assignment and not at declaration. Speaking of types, Oz introduces 2 kinds of numbers: exact numbers called INTEGERS and approximate numbers also called FLOATING POINTS. Both sets of numbers are completely independent and no automatic conversion can be achieved between them.

According to its syntax, Oz is definitely not a C-like language. Its syntax is more inspired by languages like Prolog, SmallTalk and so on. In fact, Oz syntax was carefully designed so that many paradigms do not interfere with each others. Peter Van Roy insists on 4 main differences in Oz syntax:

**Identifiers** always start with a uppercase letter.

**Procedure and function calls** are surrounded by braces: { }.

**Local identifiers** are introduced by: local <identifiers> in <statement> end.

**Variables** must respect the SINGLE-ASSIGNMENT RULE.

According to early studies, most programming bugs in Oz are introduced because of syntax errors. Therefore, it is important for students to understand and assimilate these 4 differences.

## 2.3 Functions

FUNCTION is a computing key structure. Indeed, programmers can use functions to avoid writing the same things over and over, maybe for different values. So, 4 last concepts can be introduced to complete the current functional paradigm:

**Function** It defines a program code to execute. It corresponds to another kind of value in memory. Indeed, the program code of a function is stored as a variable in memory. Therefore, each function has its own identifier.

**Function composition** Ability of a function to call another one such that it is possible to define a function in terms of other functions. Composition is a key ability for building large systems: each function is a layer built on top of another one.

**Function recursion** Ability of a function to call itself. Recursion is a key ability to build complex systems: it is possible to divide complex problems into smaller sub-problems.

**Conditional statement** Ability to change the execution depending on the value of a calculation.

These final concepts complete the functional paradigm. Indeed, students can now calculate with numbers, define functions, apply function composition and recursion, and use conditional statement. **These concepts are the key basis for every programming language.** We can say the functional paradigm allows programmers to build a TURING COMPLETE MACHINE because these concepts have the ability to do anything other computer can do.

Starting with a fundamental paradigm like functional programming is a very efficient way to approach many more paradigms. Indeed, we just need to add one or a few concepts to enter a new paradigm. This is the key idea behind the course.

## 3 Recursion, loops and invariant programming

### 3.1 Introduction to invariant programming

If a programmer combines both a RECURSIVE call and a CONDITIONAL STATEMENT, his function will act like a LOOP. In this case, the function body corresponds to one iteration of the loop. And loops are a special case of recursion called TAIL RECURSION where the recursive call is the last operation in the function body.

**Loop** A calculation repeated until a condition is satisfied to achieve a result.

In order to program correct and efficient loops, programmers may need to learn a general technique called INVARIANT PROGRAMMING. Indeed, loops can be difficult to get exactly right and a set of good practices can be useful to achieve correct loops. Invariant programming is a very important technique because it applies to both the declarative and imperative paradigms! So, this single technique allows to understand many paradigms, which is exactly the course purpose.

In the next sections, will be developped new notions related to invariant programming: SPECIFICATIONS of a function, INVARIANT of a loop, ACCUMULATOR and the PRINCIPLE OF COMMUNICATING VASES.

### 3.2 Principle of communicating vases

The goal behind the so called PRINCIPLE OF COMMUNICATING VASES is to present better ways to implement recursive functions. The concept of INVARIANT is very effective to find these *better ways*. Because an invariant is a mathematical formula, a new function can be derived from it. This new function will correspond to a new way to solve the initial problem.

**Invariant** A formula that is true at every stage of the execution.

In order to find this invariant, programmers should learn how to apply the principle of communicating vases. The principle does speak for its name: one variable should decrease, while another one should increase. Then, the invariants should be made of 2 variables at least. The first variable should decrease at each iteration of the loop. And in order to keep the formula true at each stage of execution, the second variable should increase. So, the idea of communicating vases is properly applied.

The second variable which is increasing at each iteration is called an ACCUMULATOR. Indeed, the new function - the new and better way to solve a problem - consists of an extra variable in which the result is computed and improved after each iteration.

**Accumulator** An extra argument in which the result is calculated piece by piece - we say it is accumulated.

In conclusion, invariant programming relies on the PRINCIPLE OF COMMUNICATING VASES to develop better INVARIANTS. And this principle relies itself on the notion of ACCUMULATOR.

## 3.3 Tail recursion

Invariant programming introduces the concept of loops. Loops can be programmed with TAIL RECURSIVE functions. In fact, it is proved that tail recursion is the most efficient technique to develop loops.

**Tail recursive function** When the recursive call is the last operation in the function body.

- Let's assume 2 functions that perform the same recursive computation.
- The first function is **not tail recursive**. It requires to store the *incomplete* results on a stack. Indeed, each call makes a part of the computation and these temporary results must be stored in memory. Once all the calculations are complete, the Oz interpretor must come back to these stored results to complete the final calculation. **A loop without tail recursion keeps the stack growing while executing.**
- The second function is **tail recursive**. In this case, the recursive call does not need to come back to previous results. Indeed, each computation is done before the recursive call - remember: the recursive call is the last operation - and the intermediate result is simply added to this recursive call as an argument. In conclusion, **a tail recursive loop has a constant sized stack and is more efficient because of its constant memory usage.**

## 3.4 Conclusion on invariant programming

- A recursive function is equivalent to a loop if it is TAIL RECURSIVE.
- To write tail recursive functions (loops), programmers should use an ACCUMULATOR.

- The accumulator can be found starting from an INVARIANT in respect to the PRINCIPLE OF COMMUNICATING VASES.
- This general technique is called INVARIANT PROGRAMMING and it is the only reasonable way to program loops.
- Invariant programming is useful in all programming paradigms. Indeed, other paradigm are build on top of this one.

## 4 Lists and pattern matching

### 4.1 Lists

Handling integers only may seem limited for programming more complex calculation. Therefore, programmers introduced a first compound data type called a LIST while developing LISP.

**List** An ordered sequence of elements. These elements can be of any type : an integer, an atom, or even another list. Indeed, lists can be defined in terms of themselves. Therefore, lists are defined recursively. Finally, lists are also values in memory just like integers or functions.

Syntactically and according to the EBNF rules, lists support the following formal definition:

- <List T> ::= nil | T ′|′ <List T>, where:
- A list is either empty (nil) ...
- ... or is a pair of one element (T) followed (′|′) by another list (<List T>)
- and <List T> represents the set of all syntactic representations of a list of elements of type T.

Oz support many representations for lists according to the principle of SYNTACTIC SUGAR.

**Syntactic sugar** Ability to define different textual representations of the same concept.

- Bracket notation: [1 2 3]
- EBNF notation: ′|′(1 (′|′(2 (′|′(3 nil))))
- Another equivalent notation: a | b | c | nil

## 4.2 Pattern matching

The most efficient way to program with lists is to apply the previous concept of TAIL RECURSION. The main idea when handling a list is to divide it in terms of its own structure. A programmer can divide a list into 2 parts: the HEAD which corresponds to the first element and the TAIL which corresponds to the list without its first element. Therefore, programmers can perform some computations with the head of the list, and then make a recursive call on the tail such that, at each call, the next element of the list is being handled.

Therefore, programmers developed a general technique to divide a variable in terms of its own structure. This technique is called PATTERN MATCHING. The idea is, starting from a list, to give a pattern the list should match. This pattern is more a *shape* the list should follow.

- **H|T** where H is the head of the list and T its tail. For example, the list is an integer H followed by another list T.
- **H1|H2|T** where H1 is the head of the initial list, H2 the head of the list (H2|T) and T its tail. For example, the list is an atom H1, followed by another atom H2, itself followed by an empty list T (nil).

**Pattern engineering** The ability to *play* around with different patterns. But pattern engineering can cause DEAD CODE: patterns that can never be reached. Students should stay aware of this issue.

## 4.3 Lists functions are tail recursive

Remember: the kernel language is the foundation of every programming paradigm because it contains the core concepts of computing. Therefore, programmers can translate their list functions into this kernel language. This translation gives them the ability to see all the intermediate results computed during the computation. Then, they can prove that lists functions are in fact tail recursive. Indeed, it can be proved that the output list is created **before** the recursive call. This call remaining the last operation of the function body proves that the list functions are TAIL RECURSIVE by definition.

In conclusion, **programmers should use tail recursion and pattern matching while handling lists** because they are very efficient techniques.

#### Why are lists functions tail recursive

Let's prove how the implementation of the Append function in Oz can be translated into kernel language:

- First, the function must become a procedure with an extra argument. Indeed, functions do not exist in kernel language.

- The extra argument will contain the result that should be returned by the procedure.
- Every intermediate result should become visible.

```
fun{Append L1 L2}
  case L1 of nil then L2
  [] H|T then H|{Append T L2}
  end
end
```

```
declare
proc{Append L1 L2 L3}
  case L1 of nil then L3=L2
  else
    case L1 of H|T then
 local T3 in
   L3=H|T3
   {Append T L2 T3}
 end
    end
  end
end
```

Indeed, the translation into kernel language proves that the output list L3 is generated before the recursive call to Append.

## 5 Higher-order programming and records

In this section are introduced two final concepts to complete the functional paradigm and its kernel language. These concepts are called HIGHER-ORDER PROGRAMMING and RECORDS.

**Higher-order programming** The ability to use functions and procedures as first-class entities in the language. For example, as inputs or outputs of other functions. This is a key ability to introduce DATA ABSTRACTION.

**Record** A data structure, a general compound data type that allows symbolic indexing - with the help of symbolic constants called atoms. Record is a useful concept for both symbolic programming and DATA ABSTRACTION.

### 5.1 Contextual environment and procedure value

Higher-order programming is based on 2 main concepts called CONTEXTUAL ENVIRONMENT and PROCEDURE VALUE.

**Contextual environment** A little data structure in memory related to a function (resp. procedure) that stores all the free identifiers used inside this function (resp. procedure).

**Free identifiers** An identifier used *inside* the function but declared *outside* the function.

We can easily see the need of having a contextual environment. Assume an identifier is declared twice in a program with different scope and a procedure makes use of this identifier in its body. Therefore, the identifier is said to be "free" for the procedure and the contextual environment allows to know which definition can be retrieved for the given identifier.

**Procedure value** Each procedure is a value in memory, just like integers.

When a programmer declares a procedure called "Append" in his program, the whole procedure (both its signature and body) is in fact bound to a variable called "Append" in memory.

## 5.2 Higher-order programming

Higher-order programming is a key concept in this course. Indeed, it is an important concept for building large systems based on data abstraction in layers or to achieve encapsulation. These are ideas that programmers hear all the time when programming. But why is higher-order programming so important ?

First, because **procedures (and functions) are values**, we can pass them as inputs to other functions and return them as outputs, just like integers! This introduces a concept called the ORDER of a function:

- A function whose inputs and outputs are not functions is said to be FIRST ORDER.
- A function is said to be ORDER N+1 if its inputs and outputs contain a function of maximum order N.
- For example, a function with a first order function as input is said to be a second order function.

Based on this definition of order, we can define a few more properties:

**Genericity** When a function is passed as an input.

**Instantiation** When a function is returned as an ouptut.

**Function composition** When a function takes 2 functions as input and return their composition.

**Encapsulation** The ability to hide a value inside a function. The most trivial example is to define a function called Zero which only returns the integer 0.

### 5.3 Records

Record is the last concept to study for completing the functional paradigm and its full kernel language. It is a key concept to make things such as data abstraction much more readable.

**Atom** A symbolic value - or constant - corresponding to a sequence of lowercase letters and digits that starts with a letter. It is also any sequence of any characters delimited by single quotes.

**Record** A record has a label - its name - and groups a set of values into a single compound value. Each record has a fixed number of values and each value can be accessed directly through the dot operation. Indeed, each value is stored as a pair of field name and field value separated by a colon.

**Label** An atom.

**Field names** Atoms or integers.

**Field values** Can be any value.

**Width** The width of a record is the number of values that it groups.

**Arity** The width of a record is a list of its field names.

In fact, it is *the* general compound data type used to build many other compound data types. Indeed, **it is the only compound data type in the kernel language**! It keeps the kernel language simple. Let's prove that the previous compound data types can build on top of our record definition:

**Atom** A record whose width is 0, a record with a label but without any grouped value.

**Tuple** A record whose field names are successive integers starting with 1. Note that fields declared without field names are automatically bound to successive integers starting with 1. it is another form of syntactic sugar.

**List** A recursive data type built on 2 kinds of records: nil and H|T. In fact, the following syntactic sugar is applied: $'|'$(1:H 2:T).

# 6 Trees and computational complexities

## 6.1 Trees

In this section are introduced TREES. Next to lists, trees are the most important data structure in computing. Trees are usually used to make calculations while maintaining a certain property. Indeed, many tree data structures are based on a global property, a property of the whole tree that must be maintained during the calculations. Therefore, trees are a good example of goal-oriented programming which is a generalization of invariant programming.

**Tree** A recursive data structure: it is either an empty tree called a **leaf** or an element and a set of trees called **subtrees**.

- Lists and trees are similar in terms of recursivity.
- However, a list is a linear structure (there is only one single sublist) while a tree is a branching structure with several potential subtrees.

## 6.2 Ordered binary trees

A first kind of interesting special tree is the ORDERED BINARY TREE which is based on both properties called ORDERED and BINARY as its name states.

**Binary** Each non-leaf tree has 2 subtrees, respectively named left and right.

**Ordered** For each tree, including all subtrees: all the keys in the left subtree are smaller than the key of the root, and the key of the root is smaller than all the keys in the right subtree.

**Ordered binary tree** A tree that has a key and a value field at each node - called information fields - and maintains both ordered and binary property.

## 6.3 Search trees

A second kind of interesting tree is called a SEARCH TREE. Furthermore, experiences shows that these trees are usually BALANCED.

**Search tree** A tree that is used to organize information and with which we can perform various operations such as looking up, inserting and deleting information.

**Balanced tree** A binary tree with both subtrees that have the same amount of leaves (with a difference of maximum 1) and these subtrees balanced themselves.

## 6.4 Goal-oriented programming

As seen in the introduction section, many tree algorithms depend on global properties and most of the work they do is to maintain these properties. For example, ordered binary tree must satisfy the global ordering condition. Despite the ease of inserting a new element in these trees, it is way harder to delete information from them.

Goal-oriented programming is widely used in AI algorithms. But these algorithms can give unexpected results because they try to maintain the global properties. In fact, goal-oriented programming is characteristic of living organisms. Indeed, thanks to this unpredictability, it is an efficient way to give a spark of life to some algorithms!

## 6.5 Introduction to computational complexities

An interesting question for a programmer is: “how performant is my algorithm ?”. To answer this question, programmers can try to analyse the execution time or the memory usage of their implementation. For example, it is easy to measure the number of seconds needed to execute an algorithm for a given input on a given platform. But this measure is not very useful because it is bound to a specific processor.

It is more interesting to see how the execution time depends on the input size in order to be able to predict this time on different plaftorms. So, programmers should look for a *function* and not a number. This function would allow them to predict the execution time despite the hardware composition. Even more interesting is to perform an ASYMPTOTIC ANALYSIS.

**Asymptotic analysis** The goal is to find a function that gives the execution time given an input size whose size *increases without bound.*

**Computational complexity** The use of asymptotic analysis to study the execution time and memory usage of programs.

By observing the computational complexities of some functions, programmers can highlight 2 common properties about asymptotic analysis:

- **Fast-growing functions are bad:**
  Once powers are included in a function, these functions quickly tend to take years to be computed for large input size. Moreover, they do not allow to solve big problems in short time.
- **Constant factors can be ignored:**
  In fact, 2 kinds of functions are part of the game: *polynomial* and *exponential* ones. The exponential functions are very bad because their performances do not improve very well in comparison to the hardware enhancements. So, in order to avoid the hardware influences in our functions, we should ignore constant factors.

## 6.6 Best case, average case & worst case

Let’s assume a programmer does not want to perform a temporal analysis based on integers only. Indeed, a programmer could have coded a function to store a list for example. Basically, this function would work as follow:

- **If the list is already sorted:** return the list right away.
- **If the list needs to be sorted:** do some computations to sort the list, then return it.

So, the temporal complexity might be different for different input lists ! Therefore, there is a need to compute some kind of average. There are 3 standards ways to do this:

**Best case** Takes only inputs of size n with smallest time.

**Worst case** Takes only inputs of size n with largest time.

**Average case** Takes inputs of size n according to some probability distribution (which must be given).

Therefore, programmers can highlight 3 scenarios when proceeding to a temporal analysis: a best case (e.g. when the list is empty), a worst case (e.g. when the list is not ordered at all) and an average case. These scenarios correpond to computing a lower bound, an upper bound, or both lower and upper bound. Then, we can introduce 3 new mathematical notations:

**Big O(n)** For upper bound, or worst case.

**Big Omega(n)** For lower bound, or best case.

**Big Theta(n)** For both lower and upper bound, or average case

**Asymptotic equivalence** When a function has the same lower and upper bounds.

## 6.7 Spatial complexity

Temporal complexity - which is about measuring execution time - is not the only way to scale the performance of a program. In fact, SPATIAL COMPLEXITY is also important and can be measured by 2 means:

**Spatial complexity** Memory usage measurement for a given input size.

- **Active memory active(n,t):**
  A first way to measure spatial complexity is to count the total number of words in use by the program at time t, for a given input size n. It can be useful to know the memory usage at a particular time. Indeed, different functions do not need the same amount of memory during the program execution, and they need memory capacity at this exact time, and not later.

- **Memory consumption consume(n,t):**
  A second way to measure spatial complexity is to count the number of words allocated per second at time t, for a given input size n. It can be useful to see how much temporary data structures are used by a program during its execution.
  As an analogy, active memory corresponds to a human weight (in kg) whereas memory consumption corresponds to how much this human eats (in kg/day).

## 6.8 Intractable problems and the class NP

In the previous sections were developped some ways to apprehend performance. Let's think more about it: how can we improve performance ?

- **New processor hardware** in respect to Moore's law: the density of integrated circuits doubles around every 2 years.
- **Program optimization** by developing faster but more complex algorithms eventhough optimization has its own limitations.
- **Intractable problems and P=NP question:** some problems seem to be inherently time consuming.

Basically, some problems seem to take a lot of time to be solved by a computer because there is no easy way to get around it. Therefore, the temporal complexity of the algorithms solving these problems are exponential. This introduces the class of NP PROBLEMS for NONDETERMINISTIC POLYNOMIAL TIME. Indeed, it is easy to *verify* a solution in a polynomial time, but it is way harder to *find* a solution from scratch !

The fundamental question then becomes: "Is P=NP ?", or in other words : "Is finding a solution as easy as verifying a solution ?". Note that some NP problems have the property that if an efficient algorithm can be found for them, then it is possible to derive an efficient algorithm for all problems in NP. These *hardest problems* are said to be NP-COMPLETE. They are often encountered in practice.

## 6.9 Conclusion on performance

Both execution time and memory usage are very important properties of a program to measure its performance. Asymptotic analysis should be performed to evaluate these measurements. Moreover, practical performances are influenced by 3 aspects : hardware performances, program optimization and fundamental properties of the problem (NP).

# 7 Correctness and semantics

## 7.1 Introduction to semantics

In order to write correct programmes and to understand other people's programmes, a good programmer should have a deep understanding about how a language works. The goal of this section is to learn language SEMANTICS. There are many things a programmer can do by using semantics: making sure a program is well-designed, explaining the program of other ones, understands how a program manages memory, or even ensures that a program is correct. In this section, it is actually the verification, the correctness of a program that will be interesting. A programmer can build his thinking on 3 pillars to verify the correctness of a program:

1. **Specification** Describe what we want, a mathematical formula for example.
2. **Program** What we have written in a specific programming language.
3. **Semantics** Connect the specification and the program to prove that the code executes according to what we want. Then, we can prove our program is satisfying our specification.

   **Semantics of a programming language** Or formal semantic, or mathematical semantic, is a completely precise explanation of how programs execute. It can be used to reason about program design and correctness.

Both programmation and specification have been studied previously. Therefore, this section approaches semantics only. There are 4 fundamental ways to approach semantics:

1. **Operational semantics:** Explains a program in terms of its *execution*, its operations on a rigorously defined abstract machine.
2. **Axiomatic semantics:** Explains a program in terms of properties, as an *implication*: if certain properties called preconditions hold before the execution - they are axiomatic -, then some other properties called postconditions will hold after the execution.
3. **Denotational semantics:** Explains a program as a *function* over an abstract domain, which simplifies certain kinds of mathematical analysis of the program.
4. **Logical semantics:** Explains a program as a *logical model* of a set of logical axioms, so program execution is actually deduction: the result of a program is a true property derived from the given axioms.

This section develops the OPERATIONAL SEMANTICS because it has the advantage to work for all paradigms. Indeed, programs have to run one day. This is the most general technique to approach semantics. Operational semantics can be divided in two parts: the KERNEL LANGUAGE - in which the program can be translated - and the ABSTRACT MACHINE - to execute the translated program -. The first part was already studied in the previous sections. Then, this section will mainly focus on the abstract machine operation.

## 7.2 Abstract machine

The abstract machine brings several new concepts to our study:

**Single-assignment memory** Set of variables and the values they are bound to.

**Environment** Set of link between identifiers and variables in memory.

**Semantics instruction** A pair of instruction and its environment.

**Semantic stack** A stack of semantic instructions to be executed.

**Execution state** A pair of semantic stack and the single-assignment memory.

**Execution** A sequence of execution states.

In practice, both the initial memory and the initial environment start empty. The semantic stack is filled with the program to be executed. While the semantic stack is not empty, the abstract machine will pop the instruction at the top of it and execute it according to its semantics rule. Both the memory and the environment will be modified if necessary. That is how the abstract machine does work in practice.

### 7.3 Semantics rule

A programmer must first study the SEMANTICS RULE bound to every kernel language instruction in order to know how to execute a piece of code in the abstract machine.

**Semantics rule** Describes how a kernel language instruction should be executed in the abstract machine. e.g.: how an instruction modifies the memory, when to push/pop an environment,...

**Variable creation** How a variable is created in a local instruction.

**Sequential composition** How sequential instructions are split into the semantic stack to be executed one after the other.

**Conditional** How a if ... else is managed in the abstract machine.

**Assignment** How a variable is bound to a value in memory.

**Procedure definition/call** How a procedure is defined and called in terms of memory.

## 8 Conclusion on Louv1.1x

The course was organized in 2 paths. The first one was about operations that developpers can do while programming such as invariant programming, higher-order programming and a full kernel language overview. The second path was about data structure on which we can do these operations such as lists, trees and records. Finally, both paths met up to introduce a brief overview of semantics and performance analysis.

Appendix E

# Louv1.2x - Paradigms of Computer Programmings: Abstraction and Concurrency

## 1 Introduction

### 1.1 Motivations and roadmap

Programmers need to introduce new essential concepts for building large problems that are part of the real world. Indeed, the real world is *complex* and *changing*, but also made of *independent activities*. Based on the FUNCTIONAL PROGRAMMING and its full kernel language studied during Louv1.1x, this course develops 4 new programming paradigms:

1. **Object-Oriented Programming** based on STATE (CELLS), DATA ABSTRACTION, POLYMORPHISM and INHERITANCE. Louv1.2x also introduces Java in order to better understand OOP. However, this master thesis only focuses on Oz and the Java part is willingly skipped.

2. **Deterministic Dataflow** based on CONCURRENCY (THREADS), STREAMS, AGENTS and the lack of RACE CONDITIONS.

3. **Multi-agent Dataflow** based on DETERMINISTIC DATAFLOW with PORTS and NON-DETERMINISM where it is truly needed.

4. **Active objects** combining both OBJECT-ORIENTED PROGRAMMING and MULTI-AGENT DATAFLOW.

To understand and manipulate these new paradigms, programmers should study a set of new concepts. Louv1.2x is about studying these new principles.

### 1.2 Practical organization

Louv1.2x is organized in 7 consecutive lessons. Each lesson is divided in one or more video produced by the professor Peter Van Roy. However, the 6th lesson is taught by the professor Seif Haridi that developped the Oz language along with Peter Van Roy.

The lessons are interleaved with small exercises. Some exercises are graded by the INGInious on-line grader. These exercises are accounting for 50% of the final grade. The remaining 50% must be completed thanks to the final exam. During the final exam, students have *only one* try per exercise on INGInious.

The evaluation and certification commodities are similar to Louv1.1x.

## 2 State

### 2.1 Implicit state

In the functional paradigm, there is no notion of *time*. Indeed, functions are defined in terms of mathematics and they cannot change once they are defined. Furthermore, a function cannot observe the execution of another one, but only its final result.

However, in the real world, there is both *time* and *change*. Living organisms do grow and learn. A first solution to model both time and change is to understand asbtract time. This notion can result in the concept of implicit state. We say these states are *implicit* because the language does not need to be changed. Indeed, the notion of change is purely observed in/by the programmer's head.

**Abstract time** A simplified version of time that keeps the essential property that we need: modeling change.

**State** A sequence of values calculated progressively, which contains the intermediate results of a computation.

**Implicit** Because the language did not change, we are still in the functional paradigm, it is only our programmer's mind that started to observe the execution of the function step by step.

In fact, programmers would like the program *itself* to observe changes, and not only the programmers themselves. Therefore, they should leave the functional paradigm by adding some new concepts, new ways of thinking and programming.

### 2.2 Explicit state

The objective for programmers is to extend the kernel language to make state *explicit*. Explicit, because we will modify the current language to add the concept of state to it.

In practice, explicit state is represented by the concept of CELL by programmers. Cells do respect the token equality in opposition to the concept of structure equality (for lists, records,...).

**Cell** A cell is a box with an identity (a constant, its name or address) and a content. This content can be modified: we can observe explicit change.

**Structure equality** 2 structures with the same values created separately are equal.

**Token equality** 2 cells are equal if they are the same cell, therefore 2 cells created separately are always different.

## 2.3 Semantics of state

Even though the kernel language was extended to add the concept of cell, the abstract machine is still not aware of these changes. Therefore, it should be extended too. In order to achieve this, 2 stores must be added to the abstract machine. Students are already comfortable with the first one which corresponds to the concept of SINGLE-ASSIGNMENT. The second store contains cells which are related to MULTIPLE-ASSIGNMENT.

**Single-assignment store** Immutable store that contains variables.

**Multiple-assignment store** Mutable store that contains cells.

**Cells in the store** A pair of 2 variables. The pair itself (the link between both variables) is stored in the multiple-assignment store. The variables are stored independently in the single-assignment store. The first variable cannot be changed, it represents the name of the cell, a constant denoted by a Greek letter in the store. The second variable corresponds to the content of the cell and can be swapped with another one in order to modify the cell's content. Indeed, changing the cell's content means changing the second variable of the pair.

One of the key ideas behind Louv1.1x is that the functional paradigm can be easily extended to approach many paradigms. By adding the concept of CELL and MULTIPLE-ASSIGNMENT, programmers managed to get out of this basic paradigm. The resulting paradigm is called IMPERATIVE PARADIGM.

**Imperative paradigm** Combination of functional programming and the concept of cell such that programmers can explicitly add growth and change in their programs.

## 2.4 State and modularity

The goal of Louv1.2x is to study the tools for building large and complex software systems. EXPLICIT STATE (CELLS) is a key tool to enable MODULARITY which is an important property when building large and complex programs.

**Modular** A program is *modular* with respect to a given part if this part can be changed without changing the rest of the program.

But modularity is not always easy to deploy efficiently. Therefore, choosing between functional and imperative paradigm should be thought carefully.

- **Functional paradigm**:
  1. A component never changes its behaviour. If it works once, it works for all the time.
  2. Updating a component often means changing its interface. Therefore, many other components using it might become wrong.
- **Imperative paradigm**:
  1. Updating a component does not require to modify its interface thanks to explicit state.
  2. However, the behaviour of the component might change and break some other component using it.

## 3 Data abstraction

DATA ABSTRACTION is the main organizing principle for building large and complex software systems such that many parts can be organized in layers or in compartments and do not interfere with each others. It allows to program the *complexity* of the real world. The main idea behind DATA ABSTRACTION is called ENCAPSULATION.

**Data abstraction** A part of a program that has an inside, an outside and an interface in between.

**Inside** Is hidden from the outside such that all the operations on the inside must pass through the interface.

**Interface** A set of operations that can be used according to certain rules.

**Encapsulation** Fact of enforcing the separation between inside and outside.

Encapsulation brings many assets to programmers:

- It *guarantees* that programmers can interact with the inside only through well-defined operations - remember the notion of INTERFACE.
- It *reduces the complexity.* Indeed, programmers do not have to know the internal implementation but only the interface in order to make use of a component.

- It *enables programming with large teams* in which each developer is responsible for implementing, maintaining and guaranteeing the internal behaviour of its own components. The other programmers only interact with the interface of these components and do not modify anything to their internal implementation.

There exist 4 kinds of data abstraction in practice:

1. **Object** Represents both a value and a set of operations grouped together in a single entity.
2. **Functional object** Immutable object such that invoking an object returns another object with the new value.
3. **Abstract data type** Consists of a set of values (constants) and a set of stateless operations.
4. **Stateful ADT** Uses both a cell and a secure wrapper. In practice in C or for static attributes in Java.

# 4 Object-oriented programming

## 4.1 Introduction to OOP

OBJECT-ORIENTED PROGRAMMING - also abbreviated OOP - is the second paradigm studied in Louv1.2x. It relies on 3 new principles. The first one is DATA ABSTRACTION as seen previously. The others one are POLYMORPHISM and INHERITANCE.

## 4.2 Polymorphism

POLYMORPHISM is an important principle in OOP. In fact, both objects and all kinds of data abstractions do support polymorphism in OOP. Polymorphism allows to compor-tamentalize responsibilities such that a design decision only imply a small part of the program.

**Polymorphism** An operation is polymorphic if it works correctly for arguments of different types.

**Responsibility principle** A responsibility should be concentrated in one part of the program such that a polymorphic message is understandable by many objects and each object knows how to react upon its own responsibility.

## 4.3 Inheritance

In practice, many data abstractions are very similar. For example, a multiset is a collection with no defined order, and a sequence is a multiset with a total order. Because of these similarities, programmers are often tempted to duplicate code accross their programs. But INHERITANCE was invented to avoid these inconvenient duplications. It is an important idea to avoid redundancy by building data abstractions incrementally. For example, a sequence can *inherit* from the basic properties of a multiset, then adds the total order property to it. Therefore, inheritance is very useful to avoid code duplication and to improve code reusage.

However, inheritance should be used carefully and programmers should be aware of 2 general rules:

- **Alway prefer composition to inheritance**:
  By default, a programmer should not want his classes to be extensible by inheritance.
- **Always follow the substitution principle when using inheritance.**

**Composition** When an object refers to another object in one of its attributes such that the object referred is accessed through its usual interface.

**Substitution principle** When a class A extends a class B, A cannot modify the ground basis of B. We say that A is a *conservative extension* of B.

## 4.4 OOP objects

Despite some important principles like inheritance and polymorphism, the main feature about object-oriented programming... is OBJECT indeed. OOP objects bring some interesting properties for programmers.

**Object** A data abstraction encaspulating both values and operations. Values are called *attributes* and are hidden from the outside. Operations are called *methods* and are visible from the outside as an interface. Objects are based on a simple data abstraction extended with dispatching and instantiation.

**Dispatching** Instead of having access to multiple procedures directly, objects are reduced to a single entry point which receives a message such that, when we call the object, the entry point reads the message and chooses the right method to execute.

**Instantiation** Ability to create multiple instances of an object such that each object has the same methods, but has its own states (it has its own cells as values). Indeed, each object instance is independent of others.

## 4.5 Class

DATA ABSTRACTION, DISPATCHING and INSTANTIATION are important properties of OOP objects. But the most interesting one is surely the concept of CLASS. Classes are very useful data abstractions because they bring many improvements to objects:

- A class *guarantees the objects are constructed without error.* Indeed, their syntax is verified.
- A class *improves the readability* by using concepts at the level of objects. It does not use concepts proper to procedures of cells that are lower level concepts, more basic and generalistic.
- A class *improves overall performance* if the system can find a better way to implement the object.

**Class** A specialized syntax to define many objects with the same behaviour. These objects are said to be "*of the same class*". In memory, classes are values. They are stored as a record that groups both attributes and methods definition.

## 5 Deterministic dataflow programming

### 5.1 Concurrency

The world is naturally *concurrent* because it is made of independent activities. Many examples shows computing is also highly concurrent: distributed systems, operating systems, processes,...etc. However, the paradigms studied until now do not support CONCURRENCY. There is no way of *communication* from one concurrent activity to another and there is no way of *synchronization* between concurrent activities.

**Concurrency** A property of systems that are made of activities that progress independently. It allows to model independent activities in a program.

To support concurrency, a new paradigm should be introduced. Many paradigms do support it but concurrency is quite complex to program. Therefore, Louv1.2x only introduces one of them called DETERMINISTIC DATAFLOW. Indeed, this paradigm is quite close to the functional paradigm that was already studied in Louv1.1x - even though it is truly concurrent in opposition to functional programming.

**Deterministic dataflow** A simplified form of concurrency that always gives the same outputs for the same inputs. Activities are running independently and support unbound variables. Indeed, if they encounter an unbound variable, they enter in a waiting state such that another activity can bound this variable later which would make the waiting activity to resume its execution

### 5.2 Threads

Speaking of *independent activities*, let's define them and enumerate their properties. In computing, independent activities are also called THREADS. For example, each call to the Oz function *Browse* instantiates in fact a new thread in execution.

**Activities (Thread)** A sequence of executing instructions.

1. **Sequential**: each thread is executed sequentially as any other parts of a program.
2. **Independent of other threads**: there is no order of execution defined between different threads, they're all executed using an INTERLEAVING SEMANTICS such that one thread is in execution at a time and the system guarantees that each thread will have a fair-share of the computational capacity of the only processor.
3. **Shared variables** are used for communicating between threads.

**Interleaving semantics** Assuming all the threads share the same processor, the semantic rule can be formulated as follows: "each thread has its own semantic stack with its own set of instructions and shares the memory with every other existing stacks". Therefore, a semantic stack with the current environment should be instantiated every time the abstract machine does encounter a new thread.

## 5.3 Non-determinism

In a *sequential* program, execution states respect a TOTAL ORDER property. However, in a *concurrent* program, only the execution states of the same thread respect this total order property. The execution states of the complete program respect in fact a PARTIAL ORDER property such that many executions are compatible with this partial order and the SCHEDULER chooses one of these potential scenarios during the actual execution. In fact, there is some kind of NON-DETERMINISM in the deterministic data flow paradigm that was introduced. Let's define these new terms:

**Total order** When comparing any 2 execution states, one must always happen before the other.

**Partial order** When comparing any 2 execution states, there might be no order between them (either may happen first).

**Determinism** A deterministic program always gives the same output for the same input.

**Non-determinism** Ability of a system to make decisions that are visible when running a programme such that these decisions can vary from one execution to another.

**Scheduler** The part of the system that decides at each moment which thread to execute.

And NON-DETERMINISM brings of course its set of issues. For example, using both THREADS and CELLS at the same time in a program can lead to severe dysfunctions. Therefore, non-determinism must always be managed carefully in this case.

However, **deterministic dataflow supports only threads and not cells**. Therefore, threads can only synchronize themselves with single-assignment variables. So it is its main advantage: the non-determinism of the scheduler does not affect its final result ! In fact, one can say that deterministic dataflow has "NO OBSERVABLE NON-DETERMINISM" because it has no race condition. But that does not mean the scheduler is deterministic ! Indeed, **it is non-deterministic**, but we can't observe it directly because the results are always the same, whatever non-deterministic choices are made by the scheduler.

## 5.4 Concurrency transparency

The main reason to learn about deterministic dataflow in order to introduce a concurrent paradigm is the fact that it is very close to the functional paradigm. Indeed, the CONCURRENCY TRANSPARENCY of the deterministic dataflow is a key property for programming concurrent programs that do work all the time.

**Concurrency transparency** Adding threads to a functional program does not change its final result.

In fact, including threads in a functional program makes it more incremental: it will calculate things piece by piece. But its final result will never change because of its inherent properties:

1. **A functional program**: By definition, always outputs the same result for the same inputs.
2. **No observable non-determinism**: Including threads but no cell only changes the order of execution, so when the result is calculated but not how.

# 6 Multi-agent dataflow programming

## 6.1 Streams

In order to achieve concurrency in deterministic dataflow, threads share the variables in memory. Therefore, they can communicate between them by using these shared variables. An efficient way of communication between 2 threads is to make use of a STREAM as a communication channel. Therefore, it is possible to implement a PRODUCER/CONSUMER programme in which the threads are communicating through a stream in a *pipeline* fashion.

**Stream** A list that ends with an unbound variable such that this unbound variable can be used to extend the list as long as necessary. If the unbound variable is bound to "nil", then the stream will be closed.

**Producer** A thread that generates a stream of data.

**Consumer/Transformer** A thread that reads this stream of data and performs some action with it (it can *transform* the stream for example).

**Pipeline** One kind of multi-agent program where each agent receives a stream, makes a part of the work and outputs a new stream for the next agent. The first agent is usally called PRODUCER, the intermediate ones are called TRANSFORMERS or CONSUMERS.

## 6.2 Agents

In the previous section, was introduced a first kind of multi-agent program called a PIPELINE. There are in fact 2 main multi-agent architectures encountered in practice:

**Client-Server** Where a server and many clients are running independently. The server provides some services - it receives messages and replies to them - and the clients know the server address such that they are able to ask for some of its services.

**Peer-2-Peer** A very similar architecture to client-server, except that every client is also a server - we say it is a peer. Both can communicate by receiving, sending and replying to messages.

In fact, clients, servers and peers are called AGENTS, or actors. Each of these agents has an identity, receives and processes messages and replies to them. These abilities have to be casted in our programming paradigm. Let's introduce/recall some concepts:

**Identity** or address, corresponds to the notion of PORT.

**Messages** are related to any DATA STRUCTURE. A message can be a record or even a procedure.

**Receiving messages** can be achieved by using a STREAM of messages such that new messages can be appended one by one at the end of the stream.

**Replying to messages** can be achieved by using a DATAFLOW VARIABLE in the received message such that the receiving agent can compute his answer and bind the unbound variable afterwards.

Programmers already know many data structures and how to use streams and dataflow variables. One remaining concept should be studied in order to make a correct implementation of a multi-agent program. This concept is called PORT.

## 6.3 Ports

In order to represent the identity of an agent, developpers made up a new notion called PORT which is in fact a "named streams". 2 kinds of agents can be built using ports.

**Port** A pair consisting of a name and the tail of a *message stream* such that we can send messages to the stream by using the port's name. The tail of the message stream is an unbound variable that can be bound to the next message.

**Stateless agents out of ports** Agents that will receive messages and will respond to these messages, but that do not have any state.

**Agents with state** Agents that maintain a state such that they have a state-transition function which models the evolution of the state: the current state and the received message both influence the next state.

## 6.4 Sending messages

There exist 3 properties about sending messages with ports:

**Asynchronous sending** The sender continues immediately after sending his message. Indeed, the sender does not know when the message will be processed by the receiver. Very useful when the sender does not need an answer to continue his execution.

**Ordered per thread** Multiple sends in a thread are executed in the sequential order in which they are defined.

**Not ordered from multiple threads** Multiple sends in different threads can be executed in different orders depending of the scheduler choices. **For the first time, non-determinism can be observed!**

## 6.5 Replying to messages

There are 2 common ways to reply to a given message when using ports:

**Traditional view** An agent includes his own port when sending a message such that the receiver can send back his answer to this port.

**Asynchronous reply** An agent includes a dataflow variable when sending a message such that the receiver can bind this variable with his answer. The sender will stop his execution when he encounters this unbound variable later in his program. Finally, he will resume his execution when the receiver will have bound the variable with a final answer.

## 6.6 Agent protocols

This subsection goal is to create higher-level abstractions for communication by observing the existing patterns of message exchange. These patterns are defining PROTOCOLS for exchanging messages. There are 2 main protocols that programmers can observe with multi-agent programming: BROADCAST PROTOCOL and CONTRACT NET.

**Protocol** Set of rules for sending and receiving messages.

1. **Broadcast protocol**: Rule is simple, each agent just send a message M to all the agents in the system. For this purpose, a BROADACASTER agent is implemented such that any agent that wants to broadcast a message sends a broadcast request to this broadcaster agent. This agent is also responsible for keeping up to date the list of known agents in the system.

2. **Contract net**: The goal of contract net is to choose one agent satisfying a certain criteria among a group of agents. Then, the idea is to query a set of providers such that all providers send their information back to the client and this client can choose the most suitable provider for his needs.

- **Coordinator**: The agent that is responsible for receiving requests from the users and broadcasting them in response to all slaves. This coordinator is then responsible to pick the most suitable slaves among all, inform them of his choice and finally complete the whole protocol by sending his choice to the client.
- **Slave**: An agent that receives a request message from the coordinator, sends his status to this coordinator and waits until a decision is made.

## 7 Active Objects

Louv1.2x introduces 4 new paradigms. In the previous section were already studied OOP, deterministic dataflow and multi-agent programming. Therefore, there is one remaining paradigm to study. This one is called ACTIVE OBJECTS PROGRAMMING. It is at the encounter of OOP and multi-agent programming. Indeed, active objects gathers both concepts of CLASSES and OBJECTS, and CONCURRENCY into the same paradigm. In fact, agents are not only defined in terms of PORTS that receive messages and STATE-TRANSITION FUNCTIONS that define their behaviour but also in terms of classes and objects.

In active objects programming, classes define the agent behaviour. The agent local state corresponds to the class local attributes, the possible messages correspond to the method signatures and the message handlers (the state-transition function) correspond to the method definitions. Therefore, each instantiated object of a class will have its own thread that reads a message from the stream and invokes the corresponding method to answer the received messages.

// Appendix F

# Templates INGInious

## 0.1 Mirror

In this exercise, the student is asked to implement a reverse function of an integer.

**template.py**

```
local MainMirror R in
   fun {MainMirror Int}
      local Mirror in
         fun {Mirror Int Acc}
@   @Mirror@@
           end
          {Mirror Int 0}
        end
   end

R = {MainMirror 10}
end
```

## 0.2 Prime

This exercise focuses on the use of accumulators. The recursion is therefore important.

**template.py**

```
local R Prime in
   fun {Prime N}
@   @Prime@@
   end
   R = {Prime 7}
end
```

## 0.3 Fibonacci

This exercise also focuses on the use of accumulators by making the student to use two accumulators.

### template.py

```
local Fib R in
    fun {Fib N}
        local FibAux in
            fun {FibAux N Acc1 Acc2}
@   @Fibonacci@@
            end
            {FibAux N 0 1}
      end
    end
    R = {Fib 9}
   {Browse R}
end
```

## 0.4 Flatten

The students are asked to implement their own flatten procedure. We therefore check that they do not call the official flatten in their code.

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

### template.py

```
local FlattenList S in
   fun {FlattenList L}
@   @Flatten@@
   end
   S= {FlattenList [ 1 [2 3] [4] [[5 6]]]}

end
```

## 0.5 FunAsInput

In this exercise, the student has to call a test procedure with a list of function as input. The input asked to the students is too small for us to make relevant checks.

**template.py**

```
local FunnyFunc Test in
   fun {FunnyFunc FunL L}
       case L of H|T then {FunL.1 H}|{FunnyFunc FunL.2 T}
       else nil
       end
   end

    proc {Test FunL L SolL}
    {Browse {FunnyFunc FunL L} == SolL}
   end

@  @FunAsInput@@
end
```

## 0.6 BuildMyRecord

The students are asked to complete the body of a function that builds records using the oz procedure Record.make.

**template.py**

```
local Transform R in
   fun{Transform L}
@  @BuildMyRecord@@
   end
R = {Transform [record [3 5 6] [b c d]]}
end
```

## 0.7 InfixTree

First exercise with records representing trees.

```
java -jar CorrectOz.jar input.oz -d false -nCase 1 -leafCase true
```

**template.py**

```
local Infix R Tree in
   fun {Infix Tree}
@  @InfixTree@@
   end

   Tree = btree(1:42 left:btree(1:12 left:btree(1:9 left:leaf right:leaf)
right:btree(1:21 left:leaf right:btree(1:24 left:leaf right:btree(1:31
left:leaf right:leaf)))) right:leaf)

   R = {Infix Tree}
end
```

## 0.8 SortWithTree

The student has to provide two functions with the signature and the entire body. One function has to transform a tree into a list and the other does the opposite.

```
java -jar CorrectOz.jar input.oz -d false -nIf 1 -nCase 3 -nilCase true
-leafCase true -list true
```

### template.py

```
local FromListToTree R T FromTreeToList S in

@  @SortWithTree@@
   R = {FromListToTree [42 21 12 34 9]}
   T = btree(1:42 left:btree(1:9 left:leaf right:leaf) right:leaf)
   S = {FromTreeToList T}
end
```

## 0.9 IsBalanced

The student has to provide an entire function that checks if a tree is balanced.

```
java -jar CorrectOz.jar input.oz -d false -nIf 1 -nCase 2 -leafCase true
```

### template.py

```
local NumLeaves IsBalanced R in
@  @IsBalanced@@

   R = {IsBalanced btree(1:42 left:btree(1:12 left:leaf right:leaf)
right:leaf)}
end
```