Under the supervision of Prof. Peter Van Roy

Master Thesis - A new syntax for the Oz programming language

Martin Vandenbussche - 02441500

Academic year 2020–2021

Abstract *The Oz programming language has proven over the years its value as a learning and research tool about programming paradigms, in universities around the world. It has had a major influence on the development of more recent programming languages, and has functionally stood the test of time. That being said, its syntax lacks the ability the efficiently use some modern programming paradigms; the goal of this work, building upon last year's thesis of Jean-Pacifique Mbonyincungu, is to design a brand new syntax for Oz, that will allow the language to tackle new paradigms, while remaining compatible with the existing Mozart system.*

TODO list :

- "we" versus "me/I" -> make sure to stay coherent in the whole text
- thanks at the beginning : promoter, readers, online contributors
- Definitive title (+ modify the bibliography entry to this repo to match it)
- should we include the documentation/tutorials in the appendices ? Or just a link there ? Or in the bibliography ?

Contents

1 Goal of the project and previous works

1.1 Context of the thesis and the problem to solve

The *Oz* programming language is a multi-paradigm language developed, along with its official implementation called Mozart, in the 1990s by researchers from DFKI (the German Research Center for Artificial Intelligence), SICS (the Swedish Institute of Computer Science), the University of the Saarland, UCLouvain (the Université Catholique de Louvain), and others. It is designed for advanced, concurrent, networked, soft real-time, and reactive applications. *Oz* provides the salient features of object-oriented programming (including state, abstract data types, objects, classes, and inheritance), functional programming (including compositional syntax, first-class procedures/functions, and lexical scoping), as well as logic programming and constraint programming (including logic variables, constraints, disjunction constructs, and programmable search mechanisms). *Oz* allows users to dynamically create any number of sequential threads, which can be described as dataflow-driven, in the sense that a thread executing an operation will suspend until all needed operands have a well-defined value [HF08].

Over the years, the *Oz* programming language has been used with success in various MOOCs and university courses. It's multi-paradigm philosophy proved to be an invaluable strength in teaching students the basics of programming paradigms, in a manner that very few other languages could, thanks to its ability to implement such a variety of concepts in a single unified syntax. However, it has become obvious over time that said syntax also constitutes a drawback. In particular, *Oz* has not been updated like other languages have, which is hindering its ability to keep a growing and active community of developers around it.

Building upon this observation, it was decided by Professor Peter Van Roy at UCLouvain in 2019 [TO CONFIRM, WHO and WHEN] that a new syntax would be developed for *Oz*, with the utlimate goal of including this syntax in the official release of Mozart 2. The objective behind what would later be called *NewOz* is ambitious : bringing the syntax of *Oz* to par with modern programming languages, while keeping alive the philosophy that makes its strength : giving access to a plethora of programming paradigms in a single, coherent environment. This process has started in 2020, with the master thesis of M. Mbonyincungu [Mbo20], who created a first design for the *NewOz* syntax, heavily inspired by *Ozma* and *Scala*.

This thesis continues this work by providing three major results : (a) the definition of a new, more refined version of *NewOz*'s syntax; (b) the creation of a complete, robust compiler supporting it; (c) a broad reflection on the syntax design process as a whole, the mistakes that were made during the *NewOz* project so far, and some ideas that might help future contributors to the cause.

In the following sections, we will provide an overview of what our sources of inspiration have been when designing this new syntax, which results previous works have achieved, and we will give an overview of the contributions that this thesis made to the *NewOz* project in general.

1.2 Inspirations

DESCRIBE THE TIMELINE OVER THE YEARS + FUTURE

1.2.1 Scala

Lazy capabilities
Functional programming

Lacking syntax for multi-threaded programming : *Ozma* to the rescue

1.2.2 Ozma : a Scala extension

Why this work proved that *Oz*'s philosophy could be applied in other languages and fit nicely in their syntax; How it laid the foundations of *NewOz*'s *Scala*-inspired grammar

1.2.3 NewOz 2020

Last year's work of Jean-Pacifique Mbonyincungu had as main objective to "create, elaborate and motivate a new syntax" [Mbo20] for *Oz*, by systematically reviewing a subset of the languages features and syntax elements of *Oz*. For each of these, code snippets in both *Oz* and *Scala*/*Ozma* were provided and compared. The code served as a basis for the reflection and ensuing discussion, comparing pros and cons of both existing approaches, conceiving a new one when required, and motivating the final choices being made. The process was rationalized by using a set of objective factors, allowing to rate each choice on a numeric scale in an attempt to provide the best syntax for each language feature.

This thesis has provided two main results :

- The definition of a new syntax (which we will refer to as *NewOz* 2020 in this document), as we said before; this syntax can be consulted in the appendices of

the thesis[1] in the form of an EBNF grammar. This result served as the starting point for the syntax designed in this year's work; chapter 2 describes how we covered syntax elements left untouched last year, on top of further refining other aspects.[meh]

- The writing of what we will call the "Parser", which is able to convert code written in *NewOz* to the equivalent *Oz* code. This Parser was an important step to bring [legitimacy] to the new syntax, as it allows programmers to actually use the syntax in a real-world context; however, it lacked some key functionalities present in most compilers, and wasn't very reliable. This eventually lead us to the idea that a new technical implementation of a *NewOz* compiler was necessary, as we will explain in chapter 3.

1.2.4 Other works

Used to get a sense of the philosophy behind *Oz*

- Kornstaedt 1996
- History of the *Oz* Multiparadigm Language
- Concepts, Techniques and Models of Computer Programming (does it fit here ?)

1.3 Contributions

What were the ambitions at the start ? Rappel : ceci est un chapitre d'intro

- Adaptation to JP's syntax
- *NewOz* compiler
- Community feedback - Describe process of community feedback gathering

1.4 Conclusions on the new syntax

Rappel : ceci est un chapitre d'intro

[1] See the appendix C.2 of last year's thesis [Mbo20]

2 Design principles of the new syntax

In this chapter, we will describe the general objectives we felt were important to attain with the *NewOz* syntax, as well as the characteristics that we deemed desirable for this syntax to have. We will then review the important changes that were made with respects to *NewOz 2020*, and explain the motivation behind said changes. The goal here is not to repeat what was said before by M. Mbonyincungu in [Mbo20]; the interested reader can consult his thesis for a systematic review of the syntactic changes proposed last year. We will instead focus on syntax elements that were either overlooked in that thesis, or that have been significantly modified during this year's work. Finally, we will conclude the chapter by evaluating whether this new version of *NewOz* fulfills its announced objectives, and outline potential improvements areas that we identified at that stage of the work.

TODO - go once through the whole EBNF to be sure everything is covered !

2.1 Our purpose : the big picture

The main goal of the multi-year project, as we have said before, is to create a new syntax that feels more modern to new programmers than the existing one, while keeping in the language all the functionalities that *Oz* currently has. Furthermore, this syntax should be able to integrate new concepts and paradigms in the future, in a way that is consistent with existing language features. In his thesis, M. Mbonyincungu decided to [verb] the design process around *Scala* and *Ozma*, while incorporating some elements from other languages in limited places. This has the main advantage of making the syntax very consistent from the start, provided the design process [pays attention] to only introduce elements from other languages when necessary; at any given moment, one has to ask themselves if the value provided by this new, foreign element is worth the inevitable inconsistency it will cause in the syntax, or in the general philosophy of the language.

In that regard, I think that *NewOz 2020* has been successful : this new syntax feels modern and more in par with the syntax's used nowadays, but it also feels more consistent than *Oz* in some places. Object-oriented syntax, in particular, underwent some major changes that make it way more pleasing to use. But as M. Mbonyincungu mentioned himself, *NewOz 2020* still needed maturation : it is a huge step in the right direction, but it still has flaws that need to be fixed before it could be used by online programmers or as a teaching tool. In the next section, we will go over some of those changes that we feel are worth mentioning, because they raised interesting questions and

reflections; the reader will find code examples covering those changes in appendix 5, in the form of programs written in *Oz*, *NewOz 2020* and *NewOz 2021* presented side by side.

2.2 In practice : a review of the relevant syntax elements

A first syntax element we reviewed in *NewOz 2020* was the declaration and use of variables. While the use of keywords `var` and `val` is a big improvement, and a great way to hide the behaviour of cells in Oz, the possibility that was introduced to write a semicolon ";" at the end of a line declaring variables immediately caught our attention. To quote M. Mbonyincungu's thesis, "the ";" end of line token is just a random addition inspired from *Scala* to allow those with *Scala* creating an unbound value with a peace of mind" (*sic*). This justification seems us precarious at best; not only does it go again the general idea in *Oz* that carriage returns are the preferred way to delimit statements, but it also is the only use of the semicolon character in the whole syntax. We felt like two options were available : either use this delimiter for every statement in the syntax, like in Java for example, or never use it at all. We decided to go for the second option, if only because it stays closer to the original *Oz* philosophy.

Cells in *Oz* provide a specific syntax for reading and writing their content, using respectively the tokens `@` and `:=`, whereas variables use the `=` sign. *NewOz 2020* proposed to keep this syntax for the now-called `var`s, arguing that it allows to better showcase the fundamental difference between cells and variables in *Oz*. Our take is that using the more intuitive `=` token in both places is not only aesthetically more pleasing than the dated `@` and `:=` symbols, but it also doesn't take away the teaching opportunity that *Oz*'s immutable variables represent. Indeed, the unification of the notation allows new programmers, that haven't used *Oz* in the past, to use `var`s and `val`s in an intuitive manner, with the resulting behaviour that they expect; on the other hand, students using *NewOz* can receive an explanation of the reason why `var`s are mutable, and how this is in fact implemented in *Oz* and its kernel language. For those reasons, we felt like using the more standard `=` token everywhere was a preferable solution in this case.
[Small code example]

Another element that underwent heavy changes was the way *NewOz 2020* handled lambda functions and procedures. As M. Mbonyincungu duly notes, lambdas are the same concept as what *Oz* calls anonymous functions and procedures; but in this case, we feel like the syntax proposed in *NewOz 2020* sacrifices usability, readability, and the respect of *Oz*'s philosophy for the sheer will of bringing the syntax closer to that of *Scala*. As can be seen in the "Fibonacci" example in appendix 5, *NewOz 2020*'s notation uses a `=>` like Scala or JavaScript for lambda functions. Lambda procedures, on the other hand, omit this symbol. We feel like this is not a very great way to differentiate functions and

procedures in this case, because it makes the definition of lambda procedures confusing; it is our opinion that keeping the keyword `fun` and `proc`, or rather their replacement `def` and `defproc`, would be preferable. We also think that this "*arguments* `=>` *body*" construction, while it fits vey well in *Scala*'s overall syntax, felt a little out-of-place in *NewOz*, giving the feeling that it was a syntactic sugar for something else. For those reasons, we proposed a solution that was way closer to *Oz*'s original syntax, but that still incorporates the major improvements that the new functions/procedures definition, and the revamped code blocks, represent.
[Small code example]

The syntax elements linked to object-oriented programming haven't seen many changes. The syntax for accessing class attributes has been adapted to match the changes discussed above regarding mutable variables; the motivation for this was of course to keep the language consistent with itself. The keyword `super`, used to reference the parent class, can now omit the name of said class : it is now only mandatory to avoid confusion in multi-inheritance cases. It will be up to the compiler to enforce the presence of this argument when necessary. This improvement was actually discussed by M. Mbonyincungu in his work, but it was abandoned due to the technical limitations of his Parser (see also chapter 3).
Similarly, public methods don't need to be written using an *atomLisp* anymore; this was only done due to the fact that the Parser was stateless, and thus couldn't differentiate public methods from attributes in *NewOz 2020*. Since the new compiler can now leverage a symbol table, this limitation is lifted and more "standard" function names can be used (again, see chapter 3).
[Small code example]

NewOz 2021 enforces the presence of a code block in the second part of a match structure (that is, the part after the `=>` symbol). This used to be optional in cases where the *consequence* only contained one statement or expression. However, we felt like this was kind of arbitrary, and we valued the consistency with the conditional structures - in which a proper code block with curly brackets is also mandatory - over this small quality-of-life improvement in switch-case patterns. We also feel, even though this could be a matter of personal opinions, that a code block make the code easier to read.
Similarly, the `catch` clauses often make use of pattern-matching on the caught expression. Their syntax has also been adapted to be consistent with what was discussed above, following *Oz*'s intention of making those two structures as similar as possible.

Another important improvement was the import of the complete *Oz* standard library, as described in the official Mozart documentation[1], into the compiler itself. The Parser from M. Mbonyincungu only supported a subset of pre-defined functions that were put in

[1]Mozart's online documentation provides an overview of what is called the *Oz Base Environment*, which is an extensive list of functions and procedures directly available to the programmer when writing *Oz* code. This library can be found online at [DKS08]

manually; this adaptation will allow a more convenient use of the language by developers, and is a major step towards the goal of reaching functional parity between *NewOz* and *Oz* in the future.

2.3 In the end : a self-evaluation

3 The *NewOz* Compiler : `nozc`

In this chapter, we will give a couple of definitions of concepts that are relevant to this section, and describe to the situation that*NewOz* was in, from a software perspective, at the end of last year's thesis. We will then give an evaluation of that situation, highlighting problems or areas that required the most attention. The next natural step is to describe the solution we have imagined and developed, both holistically and in technical terms. We will then conclude the chapter by providing a self-evaluation of the implementation, as well as some attention points and leads for future improvements.

3.1 A quick introduction to compilers

In programming, a compiler is a piece of software that is able to translate code written in one language, to another language. The *target* language is usually a lower-level language : the main use of compilers is to create machine level, platform-specific code that is directly executable by the computer. *C*, *Erlang* and *Rust* are examples of compiled languages. Compilers are usually designed in three main blocks : a front-end, middle-end, and back-end. [Wik21a]
The front-end typically scans the input code in a *Lexer*, recognizing keywords and known literals and storing them as *tokens*. It then proceeds with the syntax analysis, which will try to match those series of tokens to known language structures, such as statements, arithmetic operations, or method definitions. This allows for the creation of an *Abstract Syntax Tree*, which stores the program's in a structure that is not only easy to analyze and understand, but also generic enough to be compatible with the middle- and back-end. In a third step, the compiler performs *semantic analysis* on the generated *AST*, checking variable types and assignments and populating the *symbol table*, which stores the names and definitions known in the context of the program.
The middle-end of a compiler performs optimizations on the *AST* to improve the performance of the target code that will be generated in the next step. An important property of compilers is that the middle-end is typically independent of both the source language being compiled, and the target platform, thanks to the generic properties of the *AST*. A fascinating example of this property is the *GNU Compiler Collection* [Inc21a], which provides a single middle-end used in multiple front- and back-end combinations.
Finally, the back-end part of a compiler will generate the target computer code from the optimized *AST*. This code is usually machine code, specialized for a specific CPU architecture and operating system, but there are exceptions (`nozc` is one of them).

3.2 The intial situation

As M. Mbonyincungu explains in his thesis [Mbo20], creating a new syntax only makes sense if it can actually be used by programmers. This requires the creation of some kind of program able to eventually transform *NewOz* code into machine code. Two possible approaches were identified : rewriting the existing *Oz* compiler, *ozc*, or creating a NewOz-to-Oz compiler. M. Mbonyincungu decided to go with the second approach : "One of the key elements of this project is that compatibility has to be maintained with the existing Mozart system, for the official release of Mozart2. The idea of writing a new compiler has thus quickly been set aside, as it would drastically increase the time and complexity requirements of the project." [Mbo20]

Instead, that idea emerged of writing a "syntax parser" (*sic*), that would serve as a compatibility layer between the *NewOz* syntax, and the existing *Oz* syntax supported by the current version of Mozart. *NewOz* code will be translated to the directly equivalent *Oz* code, and then fed to the existing *Oz* compiler, `ozc`. Some readers might interject that this description lies closer to the definition of a compiler than a parser; for this reason, I think it is important to take the time and explicit the definition we give to each term in the context of this work.

Wikipedia defines parsing as "the formal analysis by a computer of a sentence or other string of words into its constituents, resulting in a parse tree showing their syntactic relation to each other [...]". [Wik21b] A compiler, on the other hand, is described as "a computer program that translates computer code written in one programming language (the source language) into another language (the target language)." [Wik21a] In my opinion, the program created by M. Mbonyincungu doesn't match any of those two definitions perfectly, as we will discuss later; I think it lies somewhere in between those two definitions, as a decorator to the `ozc` compiler. But to stay consistent with the vocabulary used in lest year's thesis and avoid confusion, we will refer to M. Mbonyincungu's program as "the Parser" in the rest of this document.

M. Mbonyincungu's Parser makes use of *Scala*'s Parsing Combinators library[1], which provides a syntax to match regular expressions and describe the relationship between them. The Parser used it to describe pattern-matching rules which it then applied to the *NewOz* code. Finally, the *Oz* code equivalent to each matched sentence was generated, with a great emphasis being put on maintaining the code's visual format.[2] This is important because the Parser was designed as a decorator to the Mozart compiler (which means that having code roughly at the same place will make debugging programs a lot easier), but also because it can prove useful in a teaching context in the future, when comparing the two syntax's side by side.

This "parser approach" has been preferred over a rewrite/modification of the existing

[1] See its documentation at `https://www.scala-lang.org/api/2.12.3/scala-parser-combinators/scala/util/parsing/combinator/Parsers.html` [EPF21]

[2] See sections 3.2.3 and 3.3.1 of [Mbo20]

Mozart compiler for multiple reasons, which we will comment on in the next section :

1. Because of its lower technical complexity, it would take less time to design;
2. Working on an existing codebase could have revealed unforeseen problems and limitations;
3. This approach would limit the amount of regression testing required;
4. The use of a modern technology like *Scala* would make the codebase easier to maintain and collaborate on;
5. Future extensions and modifications would be easy, thanks to the inheritance concepts embedded in the library used

M. Mbonyincungu then describes the limitations and problems identified in his approach and implementation :

6. The order in which some expressions alternations are declared in the pattern-matching code has a huge impact on the performance of the program. For example, if the code defines a statement of type A as `(p1 | p2)`, parsing `p2` in the code to compile is much more costly than parsing a statement `p1`. In practice, this results in much longer compilation time for the user, depending on the particular statements, expressions, or keywords they used. This leads to a lot of confusion from my experience, as two programs of the same syntactic complexity can have drastically different compilation time.
7. The Parser is stateless. This has a lot of implications, mainly when it comes to variable types; making it impossible, for example, to evaluate the validity of an arithmetic operation for two given arguments.

3.3 The need for something else

To explain the thought process that lead to the creation of `nozc`, we think it is important to firstly explicit our interpretation and opinion of the points enumerated above. Points 1 through 3 are very valid considerations when tackling a project of this size, especially in the context of a master thesis with limited time and a fixed deadline. In that regard, the Parser is a great solution that accomplishes its objective : allowing programmers to test and run code written using the *NewOz* syntax.
However, since this year's thesis was placed in the direct continuation of M. Mbonyincungu's work, we had a lot more time on our hands [too informal ?], which allowed us to design a solution that is more ambitious technically and, we hope, more pleasant to use. In that context, points 4 and 5 were certainly taken into account : it is now clear that the *NewOz* project's implementation will span multiple years, and it is essential to reduce the hand-over effort between maintainers to a minimum. This implies, among other things, using popular technologies, maintaining a good documentation, writing modular

and maintainable code, but also publishing it under an appropriate open-source license; these considerations are further described in the next sections.
The problem identified in point 6 is in fact inherent to the library used; as such, no amount of code optimization by the programmer could bring satisfactory results in that area. This finding alone, in our opinion, revealed the need to have a new technical approach if we were to improve the *NewOz* compiler.
Finally, the statelessness of the Parser also greatly limits the flexibility of the syntax in such a way that we could not consider it acceptable for real-world use. This further reinforced our feeling that a new approach was necessary.
Another big problem of the Parser that was mostly overlooked in last year's thesis was the limited error reporting capabilities caused by the program's inherent structure. As we said earlier, the Parser was designed to output *Oz* code in a `.oz` file, and then execute the command-line `ozc` compiler with said file in input. In practice, the Parser has limited semantic analysis capabilities, and this has two consequences. First of all, it is enough to make us hesitant to call it as a proper compiler - as we touched upon earlier, even though it obviously does a lot more than a simple parser; but more importantly, this limitation means that most errors will be caught during the second phase of the compilation, that is, during the execution of `ozc`. This has the consequence that the user will receive messages describing errors present in the *Oz* code, which might be quite different from the *NewOz* code he wrote. Moreover, we should remember that one of the goals of this approach was to make the intermediary "*Oz* step" transparent to the user, and we can't expect future programmers, who will not have worked with Mozart/Oz, to know how to interpret `ozc` error messages. Even though the Parser's output formatting does a great job at maintaining a visual equivalency between the *NewOz* and *Oz* versions of the code, some error messages will inevitably be undecipherable for the end user. In my opinion, this limitation kind of defeats the purpose of making a new syntax and compiler in the first place, and is the main reason that pushed us to conceive a new solution involving a more complete compiler.

3.4 A solution : `Nozc` in details

The *NewOz* Compiler [Van21a], which we decided to call `nozc` in reference to Mozart's `ozc` utility, is a complete compiler able to transform a *NewOz* program written in a `.noz` file, into code executable using Mozart's `ozengine` command. In that regard, it does not fit the most classic definition of a compiler, as we mentioned before, since it does not generate low-level machine code, but instead translates from one high-level language to another. The current version of `nozc` runs on Windows, MacOS, and Linux, through a command-line interface.
The overall approach used by this compiler is actually the same as the one imagined by M. Mbonyincungu for the Parser : the program will ingest a `.noz` file, write the equivalent `.oz` one, and then run `ozc` with that input. However, we believe this year's approach is technically more accomplished, as it fully encompasses the 4 main phases of a classic compiler : lexer, parser, semantic analysis, and code generation, including

a limited amount of optimization. As such, it is able to produce informative, precise error messages that make debugging a *NewOz* program a lot easier, without relying on the underlying `ozc` compiler. In that regard, we believe it is a big improvement over last year's Parser, in the sense that it addresses our main criticism towards it. The ultimate goal is to be able to handle in this compiler all warnings and errors, systematically generating *Oz* code that will pass smoothly in the underlying `ozc` compiler every single time; achieving this is essential if we want to mask the internal reliance on `ozc` to the end user.
On top of its standard compilation functionality, `nozc` also provides other useful features, such as the ability to print the syntax tree of the program directly in the command-line, or to compile multiple files at a time. Additionally, a couple of quality-of-life features have been embedded, such as a robust command-line interface that will make `nozc` easy to integrate in other tools by complying to general, good-practice CLI guidelines[3]. The user also has the ability to see the intermediary *Oz* code generated during the compilation, or else [??] to personalize the logging level of the output, by using the well-known Apache's Log4j logging levels[4].

General description of the inner workings of the compiler. Do not go in ridiculous details, as the code is well documented and available. Explain modularity : ideally, most changes should simply involve modifying the JavaCC source grammar file Use an example and show its evolution when going through the compiler.

3.5 Technologies used

As said before, an important consideration when designing `nozc` was the maintainability of the project in the future. Because this project will continue for multiple years and see different maintainers, it was important to select a technology that was either widespread and well known, or easy to apprehend, to future contributors to the project. Another point of attention is the future support of the technologies chosen: again, later contributors should be able to find support and documentation easily. For the programming language itself, our choice landed on *Java*, more specifically the last version to date, JDK16. Oracle's release cycle for Java has provided a major release every 6 months since September 2017, and it is a given at this point that Java will remain relevant for the years to come.
Other tools and libraries include :

- Picocli, a framework for creating Java command line applications following POSIX conventions[5]. A decisive factor in selecting this tool, apart from its very widespread use and great documentation, is the fact that it is designed to be shipped as a

[3]More information on those practices can be found at `https://clig.dev/#philosophy` [Pa21]
[4]To be exact, `nozc` does not use Log4j, but adopted the same logging levels per convention. See `https://logging.apache.org/log4j/2.x/log4j-api/apidocs/org/apache/logging/log4j/Level.html` for a technical description of those levels and their meanings [Fou21]
[5]The online documentation for Picocli is located at `https://picocli.info/` [Pop21]

single `.java` file to include in the final application's source code. This means that upstream maintenance is not really a concern, as the source code is directly available to the programmer and can be easily be modified locally in the future, would ever need be.

- JavaCC, a powerful parser generator creating a parser executable in a JRE[6]. This tool is by far the most interesting improvement over last year's Parser. JavaCC provides a flexible and easy-to-use grammar to describe the grammar rules of the source language. This, along with its very complete documentation and wide community, means that a new maintainer should be able to quickly get a grip on [too informal ?] this part of the compiler, which is the one most [probable] to be modified in the future, as we said before. JavaCC works by reading a grammar file, written by the user, describing the lexical and syntactic grammars of the language. It then automatically generates *Java* classes describing a lexer and a parser, which can then be used to build the abstract syntax tree for valid programs, or report errors when needed. This solution saves a lot of time compared to writing a lexer and parser from scratch, with no identifiable drawbacks in our use case.
- Gradle[7], a build and packaging tool offering great documentation, regular updates and a powerful DSL, with built-in support in the most popular *Java* IDEs. It is also designed to integrate automatically in any CD/CI pipeline.
- JUnit, the best unit testing framework for *Java* programs. An additional library called System Rules[8] was used for some specific test cases.

Overall, a great emphasis has been put on making `nozc` a future-proof and maintainable tool by : (a) using popular tools that, if they are not already mastered by future contributors, can be in a timely manner; (b) using tools that are actively maintained, reducing the risk associated with legacy code; (c) selecting trusted, open-source software, with licences that make them suited for use in our context; (d) limiting the amount of external tools used, once again to reduce the risk of dependencies depreciation in the future.
The program itself is published on GitHub under the BSD license[9].

3.6 Evaluation of our approach

We are convinced that the approach we selected with `nozc` makes it a great tool for the future contributors who will continue to work on *NewOz*'s syntax in the coming

[6] An overview of JavaCC's features can be found at `https://javacc.github.io/javacc/` [VS21]
[7] Gradle's homepage is located at `https://gradle.org/` [Inc21b]
[8] This collection of JUnit [Tea21] rules allows to test programs that make use of the *System.exit()* instruction, allowing to test the correctness of the program's return codes directly from a JUnit test suite, without having to interrupt it. See `https://stefanbirkner.github.io/system-rules/index.html` [BP20]
[9] This license is available for consultation at `https://github.com/MaVdbussche/nozc/blob/master/LICENSE`

years. The modularity of the code makes it easy to add and remove language features without affecting others, while remaining flexible by making few assumptions about the language's grammar. The code is also well documented, and we strongly believe that it can serve as a stepping stone towards the creation of a complete software ecosystem around *NewOz*.

However, we have to mention limitations that we identified in our current implementation.
The main one, in our opinion, is the inability of the compiler to print the generated *Oz* code in a format that stays as close as possible to that of the source *NewOz* code. This is due to the fact that the lexer, in this particular implementation, ignores spaces and new line characters when reading the input. This comes as a disadvantage compared to last year's Parser, but it also allows for a lot more flexibility in the way the programmer is allowed to format the source code. This issue can raise some concerns, as we touched upon earlier : it implies that error messages generated by the underlying `ozc` compiler will most probably indicate an erroneous line number to the programmer. However, this problem will progressively disappear over time with the maturation of `nozc`, as more and more of those errors will get caught in the first phase of the compilation.

Another issue with of our approach lies in the fact that this compiler does not free itself from the dependency on the legacy `ozc`, which was one of our criticism towards M. Mbonyincungu's Parser implementation. A more mature compiler should be able to generate machine code directly, or at the very least code that can be executed though Mozart's `ozengine` command, by itself, without relying on another piece of software. As often seems to be the case in master theses however, time was a limiting factor; supporting machine code generation for the various existing systems would take a lot of time and effort which we simply didn't have this year.
A solution to consider could be to rely on the JVM's multi-platform capabilities, by making `nozc` output JVM bytecode, effectively removing the need for "manual" multi-platform support. However, this approach would also come with its own drawbacks and difficulties, as some programming paradigms provided by *Oz* and *NewOz* will probably be difficult to support and implement on the JVM (in particular, one would lose Mozart's support for fine-grain threads, dataflow, and failed values)[10].
Another solution would be to fork the existing `ozc` compiler and modify its front-end to accept the new syntax.[Reformulate : "plug" nozc as a front-end to ozc]

But the main area of focus for future `nozc` improvements should probably be its integration in the existing Mozart environment through its Emacs interface. The ability to compile regions of code directly from the Emacs editor is a major feature of *Oz*, that has been left aside in this current implementation. There is a lot of gains to be made here, especially from a teaching standpoint. This would probably be a massive undertaking

[10]Further reflections on this approach might benefit from reading the work of Sébastien Doeraene on Ozma [Doe03]

though, and would require some knowledge of the Emacs system in general, and Mozart in particular.

As you can see, even though we feel like this result is a significant improvement over last year's Parser, there is a lot of work to be done before the publication of a first release version of *nozc*. We are confident however in the fact that the current *beta* version is a significant first step in that direction.

4 Evaluation of *NewOz*'s syntax

In this chapter, we will describe the process we put in place to obtain a good evaluation of the syntax proposed in chapter 2. Starting with the [things] we put in place to gather feedback from various developers from both in and outside our network, we will then give a first critical evaluation of this approach for gathering feedback, and explain the reasons that pushed us to adjust it in a second phase. Finally, we will conclude the chapter by giving a broader reflection on the approach this thesis took, both when it comes to the design and the evaluation of the syntax, but also on the future we envision for *NewOz*. Our hope is that those reflections will help future contributors select the most appropriate approach in their work, in order to make *NewOz* as successful as possible.

4.1 A first approach : gathering community feedback

This chapter = general community feedback - résumé des suggestions.
Describe how we reached potential contributors, GitHub issues mgmt, repository, etc.
What was the objective with this feedback ?

"We will now describe the feedback we received on GitHub"

4.1.1

Describe changes that were proposed to the *NewOz* syntax. For each :

- Description of the change
- Motivations (previous problem, how it fixes it, philosophy of *Oz*) : personal opinion
- Implications on other existing features
- Implications in the compiler
- What did others think of it ? (probably =/= my opinion) Should we integrate their feedback ? Why/why not ?

4.2 Première évaluation et ajustement de l'approche

What was user feedback in general ? First impressions of newcomers (relevant for forging our expectations on what future students will say, for example).

Can we say this feedback met our goal described in *NewOz*'s philosophy ? Not in terms of numbers. In terms of content, we hoped for "deeper"/"higher-level" reflections. Instead, we mainly got propositions for the usage of a particular keyword or small-scope syntax modifications.
We identify two possible reasons for this discrepancy between the expected and the actual feedback.

First of all, outside users will use the language for a short amount of time before giving feedback. Granted, we can't reasonably blame them for not willing to invest hours upon hours on contributing to an open source project online, to which they dedicate some time freely. But this means that the feedback they are able to give is mainly focused on what is apparent at first glance, that is, the "vocabulary" of the syntax. Content-focused ["de fond"] reflections can only come after extensive use of the syntax, after writing different programs using various programming paradigms. In that regard, calling upon the online community to help us in a deep reflection on the [approach] for a syntax was probably an approach that was doomed to fail.

Nevertheless, the remarks we did gather raised interesting questions and will definitely be useful in the design process of *NewOz*. Relevant syntax elements from different languages were proposed, and it is clear that such proposals are essential to design a good syntax, simply because the experience of each programmer is different, and so is their knowledge and approach of what a powerful, convenient, or event fun programming syntax is.

4.3 A second approach : a broader reflection on the project itself

5 Conclusion

Résumé de l'approche, résumé des chapitres How the situation of *Oz* has evolved thanks to this works.
What did we do well, what did we miss ? (use User feedback examples)
What could future works do ? (refer to aforementioned compiler improvements, user feedback left to address)

So far *NewOz* focused on the subset of *Oz* used at UCL and presented in [VH04]. A mature version of *NewOz* should allow programmers to use the full capabilities of the *Oz* language in the new syntax, which is a necessary step if *NewOz* is to be included in the official release of Mozart 2.

Bibliography

[Doe03] Sébastien Doeraene. "Ozma: Extending Scala with Oz Concurrency". Prom. Peter Van Roy. MA thesis. École Polytechnique de Louvain, UCLouvain, 2003. URL: ???.

[VH04] Peter Van-Roy and Seif Haridi. *Concepts, techniques, and models of computer programming*. MIT press, 2004.

[DKS08] Denys Duchier, Leif Kornstaedt, and Christian Schulte. *The Oz Base Environment*. 2008. URL: http://mozart2.org/mozart-v1/doc-1.4.0/base/index.html.

[HF08] Seif Haridi and Nils Franzén. *Tutorial of Oz*. 2008. URL: http://mozart2.org/mozart-v1/doc-1.4.0/tutorial/index.html.

[Mbo20] Jean-Pacifique Mbonyincungu. "A new syntax for Oz". Prom. Peter Van Roy. MA thesis. École Polytechnique de Louvain, UCLouvain, 2020. URL: https://dial.uclouvain.be/memoire/ucl/object/thesis:25311.

[EPF21] Scala center at EPFL. *Scala Parser Combinators*. 2021. URL: https://www.scala-lang.org/api/2.12.3/scala-parser-combinators/scala/util/parsing/combinator/Parsers.html.

[Inc21a] Free Software Foundation Inc. *GNU Compiler Collection*. 2021. URL: https://gcc.gnu.org.

[Pa21] Aanand Prasad and Ben Firshman et al. *Command Line Interface Guidelines*. 2021. URL: https://clig.dev/.

[Wik21a] Wikipedia. *Compiler*. May 14, 2021. URL: https://en.wikipedia.org/wiki/Compiler.

[Wik21b] Wikipedia. *Parsing*. May 14, 2021. URL: https://en.wikipedia.org/wiki/Parsing.

Software and Tools

[BP20] Stefan Brikner and Marc Philipp. *System Rules*. 2020. URL: https://stefanbirkner.github.io/system-rules/index.html.

[Fou21] The Apache Software Foundation. *Apache Log4j 2*. 2021. URL: https://logging.apache.org/log4j/2.x/.

[Inc21b] Gradle Inc. *Gradle Build Tool*. 2021. URL: https://gradle.org/.

[Pop21] Remko Popma. *Picocli - A mighty tiny command line interface*. 2021. URL: https://picocli.info/.

[Tea21] The JUnit Team. *JUnit4 - A programmer-oriented testing framework for Java*. 2021. URL: https://junit.org/junit4/.

[VS21] Sreeni Viswanadha and Sriram Sankar. *JavaCC - The most popular parser generator for use with Java applications*. 2021. URL: https://javacc.github.io/javacc/.

Personnal contributions

[Van21a] Martin Vandenbussche. *Nozc*. GitHub repository. 2021. URL: `https://github.com/MaVdbussche/nozc`.

[Van21b] Martin Vandenbussche. *This thesis' repository*. GitHub repository. 2021. URL: `https://github.com/MaVdbussche/Master-Thesis`.

Appendices

Appendix A : *NewOz* EBNF Grammar (2021 version)

This EBNF grammar is a reworked version of the one provided in the appendices of Jean-Pacifique Mbonyincungu's thesis, removing left-recursion problems and including changes made in the syntax since then.

```
EBNF grammar for newOz, suitable for recursive descent
- Note that the concatenation symbol in EBNF (comma) is
omitted for readability reasons
Notation      Meaning
================================================================================
ϵ              singleton containing the empty word
(w)            grouping of regular expressions
[w]            union of \epsilon with the set of words w (optional group)
{w}            zero or more times w
{w}+           one or more times w
w1 w2           concatenation of w1 with w2
w1|w2           logical union of w1 and w2 (OR)
w1 − w2        difference of w1 and w2

// Interactive statements [ENTRYPOINT]
interStatement ::= statement
             | DECLARE LCURLY {declarationPart}+ [interStatement] RCURLY

statement ::= nestConStatement
             | nestDecVariable
             | SKIP
             | SEMI
             //| DECLARE statement //TODO removed bcs matched in interStatement ?
             | RETURN expression

expression ::= nestConExpression
             | nestDecAnonym
             | DOLLAR
             | term
             | THIS
             | LCURLY expression {expression} RCURLY //TODO not implemented like thi

parExpression ::= LPAREN expression RPAREN

inStatement ::= LCURLY {declarationPart} {statement} RCURLY //TODO added possibilit
             | LCURLY {declarationPart} expression RCURLY

inExpression ::= LCURLY {declarationPart} [statement] expression RCURLY
             | LCURLY {declarationPart} statement RCURLY

nestConStatement ::= assignmentExpression
             | variable LPAREN {expression {COMMA expression}} RPAREN
             | {LCURLY}+ expression {expression} {RCURLY}+
             | LPAREN inStatement RPAREN
             | IF parExpression inStatement
                 {ELSE IF LPAREN expression RPAREN inStatement}
```

```
                [ELSE inStatement]
            | MATCH expression LCURLY
                {CASE caseStatementClause}+
                [ELSE inStatement]
              RCURLY
            | FOR LPAREN {loopDec}+ RPAREN inStatement
            | TRY inStatement
                [CATCH LCURLY
                    {CASE caseStatementClause}+
                RCURLY]
                [FINALLY inStatement]
            | RAISE inExpression
            | THREAD inStatement
            | LOCK [LPAREN expression RPAREN] inStatement

nestConExpression ::= LPAREN expression RPAREN
            | variable LPAREN {expression {COMMA expression}} RPAREN
            | IF LPAREN expression RPAREN inExpression
                {ELSE IF LPAREN expression RPAREN inExpression}
                [ELSE inExpression]
            | MATCH expression LCURLY
                {CASE caseExpressionClause}+
                [ELSE inExpression]
              RCURLY
            | FOR LPAREN {loopDec}+ RPAREN inExpression
            | TRY inExpression
                [CATCH LCURLY
                    {CASE caseExpressionClause}+
                RCURLY]
                [FINALLY inStatement]
            | RAISE inExpression
            | THREAD inExpression
            | LOCK [LPAREN expression RPAREN] inExpression

nestDecVariable ::= DEFPROC variable
                LPAREN {pattern {COMMA pattern}} RPAREN inStatement
            | DEF [LAZY] variable
                LPAREN {pattern {COMMA pattern}} RPAREN inExpression
            | FUNCTOR [variable] {
                (IMPORT importClause {COMMA importClause}+)
                    | (EXPORT exportClause {COMMA exportClause}+)
                }
                inStatement
            | CLASS variableStrict [classDescriptor] LCURLY
                {classElementDef} RCURLY

nestDecAnonym ::= DEFPROC DOLLAR
                LPAREN {pattern {COMMA pattern}} RPAREN inStatement
            | DEF [LAZY] DOLLAR
                LPAREN {pattern {COMMA pattern}} RPAREN inExpression
            | FUNCTOR [DOLLAR] {
                (IMPORT importClause {COMMA importClause}+)
```

```
                    | (EXPORT exportClause {COMMA exportClause}+)
                }
              inStatement
            | CLASS DOLLAR [classDescriptor] LCURLY
                {classElementDef} RCURLY

importClause ::=  variable
                [LPAREN (atom|int)[COLON variable]
                {COMMA (atom|int)[COLON variable]} RPAREN]
                [FROM atom]

exportClause ::= [(atom|int) COLON] variable

classElementDef ::= DEF methHead [ASSIGN variable]
                (inExpression|inStatement)
            | classDescriptor

caseStatementClause ::= pattern {(LAND|LOR) conditionalExpression}
                IMPL inStatement

caseExpressionClause ::= pattern {(LAND|LOR) conditionalExpression}
                IMPL inExpression

assignmentExpression ::= conditionalExpression
                [(ASSIGN|PLUSASS|MINUSASS|DEFINE) assignmentExpression]

conditionalExpression ::= conditionalOrExpression

conditionalOrExpression ::= conditionalAndExpression
                {LOR conditionalAndExpression}

conditionalAndExpression ::= equalityExpression
                {LAND equalityExpression}

equalityExpression ::= relationalExpression
                {EQUAL relationalExpression}

relationalExpression ::= additiveExpression
                [(GT|GE|LT|LE) additiveExpression]

additiveExpression ::= multiplicativeExpression
                {(PLUS|MINUS) multiplicativeExpression}

multiplicativeExpression ::= unaryExpression
                {(STAR|SLASH|MODULO) unaryExpression}

unaryExpression ::= (INC|DEC|MINUS|PLUS) unaryExpression
            | simpleUnaryExpression

simpleUnaryExpression ::= LNOT unaryExpression
            | postfixExpression
```

```
postfixExpression ::= primary {selector} {(DEC|INC)}

primary ::= parExpression
            | THIS DOT
                variable [LPAREN {expression {COMMA expresssion}} RPAREN]
            | SUPER LPAREN variableStrict RPAREN DOT
                variable [LPAREN {expression {COMMA expression}} RPAREN]
            | literal
            | qualifiedIdentifier
            | initializer

// Terms and patterns
term ::= atom
            | atomLisp LPAREN
                [[feature COLON] expression
                {COMMA [feature COLON] expression}] RPAREN

pattern ::= {LNOT} variable | int | float | character | atom | string
            | UNIT | TRUE | FALSE | UNDERSCORE | NIL //TODO we can remove character
            | atomLisp LPAREN [[feature COLON] pattern
                {COMMA [feature COLON] pattern} [COMMA ELLIPSIS]] RPAREN
            | LPAREN pattern {(HASHTAG|COLCOL) pattern} RPAREN
            | LBRACK [pattern {COMMA pattern}] RBRACK
            | LPAREN pattern RPAREN

declarationPart ::= (VAL|VAR) (variable|pattern)
                ASSIGN (expression|statement)
                {COMMA (variable|pattern) ASSIGN (expression|statement)} //TODO why

loopDec ::= variable IN expression [DOTDOT expression] [SEMI expression]
            | variable IN expression SEMI expression SEMI expression
            | BREAK COLON variable
            | CONTINUE COLON variable
            | RETURN COLON variable
            | DEFLT COLON expression
            | COLLECT COLON variable

literal ::= TRUE | FALSE | NIL | int | string | character | float //TODO we can rem

//label ::= UNIT | TRUE | FALSE | variable | atom //TODO actually not used anywhere

feature ::= UNIT | TRUE | FALSE | atom | int | NIL //TODO not implemented like this

classDescription ::= EXTENDS variableStrict {COMMA variableStrict}+
            | ATTR variable [ASSIGN expression]
            | PROP variable

//attrInit ::= ([LNOT] variable | atom | UNIT | TRUE | FALSE) [COLON expression] //

methHead ::= ([LNOT] variableStrict | atomLisp | UNIT | TRUE | FALSE) //TODO not im
                [LPAREN methArg {COMMA methArg}
                [COMMA ELLIPSIS] [DOLLAR] RPAREN]
```

```
methArg ::= [feature COLON] (variable | UNDERSCORE) [LE expression]

variableStrict ::= UPPERCASE {ALPHANUM}
              | LACCENT {VARIABLECHAR | PSEUDOCHAR} LACCENT

variable ::= LOWERCASE {ALPHANUM}
          | APOSTROPHE {VARIABLECHAR | PSEUDOCHAR} APOSTROPHE //TODO really ?

atom ::= atomLisp
        | RACCENT {ATOMCHAR | PSEUDOCHAR} RACCENT

atomLisp ::= APOSTROPHE (LOWERCASE | UPPERCASE) {ALPHANUM}

string ::= QUOTE {STRINGCHAR | PSEUDOCHAR} QUOTE

character ::= CHARINT
            | DEGREE CHARCHAR
            | DEGREE PSEUDOCHAR
            | CHAR // TODO in this case we should send a warning during analysis th

int ::= [MINUS] DIGIT
        | [MINUS] NONZERODIGIT {DIGIT}
        | [MINUS] "0" {OCTDIGIT}+
        | [MINUS] ("0x"|"0X") {HEXDIGIT}+
        | [MINUS] ("0b"|"0B") {BINDIGIT}+

float ::= [MINUS] {DIGIT}+ DOT {DIGIT} [("e" | "E")[~]{DIGIT}+]

boolean ::= TRUE | FALSE
```

Appendix B : Lexical Grammar

```
Lexical grammar for newOz
Notation      Meaning
================================================================================
ϵ              singleton containing the empty word
(w)            grouping of regular expressions
[w]            union of \epsilon with the set of words w (optional group)
{w}            zero or more times w
{w}+           one or more times w
w1 w2          concatenation of w1 with w2
w1|w2          logical union of w1 and w2 (OR)
w1 − w2        difference of w1 and w2

// White spaces - ignored
WHITESPACE ::= (" "|"\b"|"\t"|"\n"|"\r"|"\f")

// Comments - ignored
("//" {~("\n"|"\r")} ("\n"|"\r"["\n"])) | "?"
```

```
// Multi-line comments - ignored
"/*" {CHAR - "*/"} "*/"

// Reserved keywords
//ANDTHEN ::= "andthen"
AT      ::= "at"
ATTR    ::= "attr"
BREAK   ::= "break"
CASE    ::= "case"
CATCH   ::= "catch"
//CHOICE  ::= "choice"
CLASS   ::= "class"
//COLLECT ::= "collect"
//COND    ::= "cond"
CONTINUE  ::= "continue"
DECLARE ::= "declare"
DEF     ::= "def"
DEFPROC ::= "defproc"
DEFAULT ::= "default"
//DEFINE  ::= "define"
//DIS     ::= "dis"
//DIV     ::= "div"
DO      ::= "do"
ELSE    ::= "else"
//ELSECASE  ::= "elsecase"
//ELSEIF  ::= "elseif"
//ELSEOF  ::= "elseof"
//END     ::= "end"
EXPORT  ::= "export"
EXTENDS ::= "extends"
//FAIL    ::= "fail"
FALSE   ::= "false"
//FEAT    ::= "feat"
FINALLY ::= "finally"
FOR     ::= "for"
FROM    ::= "from"
//FUN     ::= "fun"
FUNCTOR ::= "functor"
IF      ::= "if"
IMPORT  ::= "import"
IN      ::= "in"
LAZY    ::= "lazy"
//LOCAL   ::= "local"
LOCK    ::= "lock"
MATCH   ::= "match"
METH    ::= "meth"
//MOD     ::= "mod"
NIL     ::= "nil"
//NOT     ::= "not"
//OF      ::= "of"
OR      ::= "or"
```

```
//ORELSE  ::= "orelse"
//PREPARE ::= "prepare"
//PROC    ::= "proc"
PROP    ::= "prop"
RAISE   ::= "raise"
//REQUIRE ::= "require"
RETURN  ::= "return"
//SELF    ::= "self"
SKIP    ::= "skip"
//THEN    ::= "then"
THIS    ::= "this"
THREAD  ::= "thread"
TRUE    ::= "true"
TRY     ::= "try"
UNIT    ::= "unit"
VAL     ::= "val"
VAR     ::= "var"

ASSIGN      ::= "=" ok
DEFINE      ::= ":=" ok
PLUSASS  ::= "+=" ok
MINUSASS ::= "-=" ok
EQUAL       ::= "==" ok
NE          ::= "\=" ok
LT          ::= "<" ok
GT          ::= ">" ok
LE          ::= "=<" ok
GE          ::= ">=" ok
LBARROW     ::= "<=" ok
IMPL        ::= "=>" ok
AND         ::= "&" ok TODO DELETED
LAND        ::= "&&" ok
PIPE        ::= "|" ok TODO DELETED
LOR         ::= "||" ok
LNOT        ::= "!" ok
LNOTNOT     ::= "!!" ok
MINUS       ::= "-" ok
PLUS        ::= "+" ok
STAR        ::= "*" ok
SLASH       ::= "/" ok
BACKSLASH   ::= "\" ok
MODULO      ::= "%" ok
HASHTAG     ::= "#" ok
UNDERSCORE  ::= "_" ok
DOLLAR      ::= "$" ok
APOSTROPHE  ::= "'" ok
QUOTE       ::= """ ok
LACCENT     ::= "`" ok
RACCENT     ::= "ŧ" ok
HAT         ::= "^" ok
BOX         ::= "[]" ok
//TILDE       ::= "~" ok
```

```
DEGREE      ::= "ř" ok
//COMMERCAT   ::= "@" ok
//LARROW      ::= "<-" ok
//RARROW      ::= "->" ok
//FDASSIGN    ::= "=:" //skipped
//FDNE        ::= "\=:" //skipped
//FDLT        ::= "<:" //skipped
//FDLE        ::= "=<:" //skipped
//FDGT        ::= ">:" //skipped
//FDGE        ::= ">=:" //skipped
COLCOL      ::= "::" ok
//COLCOLCOL   ::= ":::" ok

COMMA       ::= "," ok
DOT         ::= "." ok
LBRACK      ::= "[" ok
LCURLY      ::= "{" ok
LPAREN      ::= "(" ok
RBRACK      ::= "]" ok
RCURLY      ::= "}" ok
RPAREN      ::= ")" ok
SEMI        ::= ";" ok
COLON       ::= ":" ok
DOTDOT      ::= ".." ok
ELLIPSIS    ::= "..." ok

// Literals
UPPERCASE      ::= "A"|...|"Z" ok
LOWERCASE      ::= "a"|...|"z" ok
DIGIT          ::= "0"|...|"9" ok
NONZERODIGIT   ::= "1"|...|"9" ok
CHARINT        ::= "0"|...|"255" ok
ALPHANUM       ::= UPPERCASE | LOWERCASE | DIGIT | "_" ok
ATOMCHAR       ::= CHAR - ("'"|"\")
STRINGCHAR     ::= CHAR - ("""|"\")
VARIABLECHAR   ::= CHAR - ("`"|"\")
CHARCHAR       ::= CHAR - ("\")
ESCCHAR        ::= "a"|"b"|"f"|"n"|"r"|"t"|"v"|"\"|"'"|"""|"`"|"ř"
OCTDIGIT       ::= "0"|...|"7" ok
HEXDIGIT       ::= "0"|...|"9"|"A"|...|"F"|"a"|...|"f" ok
BINDIGIT       ::= "0"|"1" ok
NONZERODIGIT   ::= "1"|...|"9" ok
PSEUDOCHAR     ::= "\" OCTDIGIT OCTDIGIT OCTDIGIT ok
                 | "\" ("x" | "X") HEXDIGIT HEXDIGIT ok

// End of file
EOF            ::= "<end of file>"
```

Appendix C : Some Examples

Code examples : Oz vs *NewOz*

- everytime : show Oz + NewOz2020 + NewOz2021
- lambdas (functions and procedure) - fibo example
- classic small math stuff with ifs
- OOP
- tail-recursion and lists/streams syntax

Appendix D : Documentation and tutorial

Move this as the first appendix ?