

Master thesis under the supervision of Prof. Peter
Van Roy

NewOz: Steps toward a modern 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 for 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 to efficiently use some modern programming paradigms; the goal of this work, building upon last year's thesis of Jean-Pacifique Mbonyingungu, 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.*

I would like to thank the readers of this thesis, Nicolas Laurent, H  l  ne Verhaeghe, and Martin Henz, for dedicating some of their time to this project. I would also like to show our appreciation to the people who contributed to the discussion over on GitHub, especially users pnrao and BarghestEPL for their valuable opinions. Acknowledgement is also due to my employer, Belfius Banque SA, for accommodating my modified schedule during the writing of this thesis. I am deeply grateful for their understanding and support. Finally, I want to express my gratitude to Professor Peter Van Roy for his dedication and implication in all stages of this project. His passion for Oz is beautiful to see.

Contents

1. Goal of the project and previous works	4
1.1. Context of the thesis and the problem to solve	4
1.2. Our inspirations : a brief history	5
1.3. Contributions of this thesis	7
1.4. Conclusions and the road ahead	7
2. Design principles of the new syntax	8
2.1. Our purpose : the big picture	8
2.2. In practice : a review of the relevant syntax elements	9
2.3. In the end : a self-evaluation	18
3. The <i>NewOz</i> Compiler : <i>nozc</i>	19
3.1. A quick introduction to compilers	19
3.2. The initial situation	20
3.3. The need for something else	21
3.4. A solution : <i>Nozc</i> in details	22
3.5. Technologies used	23
3.6. Evaluation of our approach	25
4. Evaluation of <i>NewOz</i>'s syntax	27
4.1. A first approach : gathering community feedback	27
4.2. First evaluation and adjustments	30
4.3. A second approach : a broader reflection on the project itself	31
5. Conclusion	32
Appendices	37
A. Appendix A : <i>NewOz</i> EBNF Grammar (2021 version)	37
B. Appendix B : Lexical Grammar (2021 version)	41
C. Appendix C : Some Examples	44
D. Appendix D : Compilation example	44
E. Appendix E : Documentation and tutorial	44

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. Its multi-paradigm philosophy proved to be a valuable 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 that a new syntax would be developed for *Oz*, with the ultimate 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*.

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. Our inspirations : a brief history

Just like spoken languages, programming languages evolve over time. The needs of the industry are in constant motion, and therefore, the offer of programming languages has to adapt constantly. The only thing that seems certain, is that a modern programming language has to have multi-paradigm capabilities. Gone are the days of *Smalltalk* or *Prolog*, which were completely designed around a specific use case and its accompanying paradigm. Today, programmers need the ability to handle heavy computational workloads, on multi-core systems in a distributed environment, with thousands of clients; and the ability to perform this in a single environment is highly valued. In that regard, *Oz* was a pioneer; never before was a language able to implement so many different paradigms, and the influence it has had over other languages is the best proof of how big a deal this was.

The observation that multi-paradigm languages are now the norm, leads us to the following question : what could the ultimate multi-paradigm language, the one to rule them all, look like ? In a darwinist way of thinking, it is probable that it doesn't exist yet, and that it never will. The only certain thing is that it will take elements from existing languages and expand upon them. This is the premise of our design process : keep the general ideas behind *Oz*, but express them through a new syntax that is closer to what modern programming languages look like.

1.2.1. Scala : a vantage point

In a lot of ways, the *Scala* syntax is the perfect example of a modern multi-paradigm language. One of the reasons it was created in the first place was to address the lack of support for functional programming in Java, while keeping its powerful object-oriented capabilities; on top of this, the huge library it inherits from Oracle's language allows it to be used in the most various of situations, and to be extended easily. Moreover, it natively supports a lot of features found in *Oz* : lazy evaluation, immutability, anonymous functions, actor model¹,...

That is not to say that *Scala* is the perfect language : it has a very steep learning curve, which may be why the language as a whole doesn't quite have the popularity we

¹In the last versions of *Scala*, the use of the *Akka* toolkit[Inc21c], written in *Scala*, is the preferred method for writing programs leveraging distributed programming

could have expected². However, it still seems overall that *Scala* is a very good point to start our journey towards a vision of a definitive multi-paradigm language.

1.2.2. Ozma : a springboard

We were not the first to take interest in *Scala* among the *Oz* community. And we were certainly not the first to notice how the language lacks some critical elements in the context of our quest.

The 2003 thesis of Sébastien Doeraene [Doe03] investigated the idea of adding the elegant and efficient concurrency capabilities of *Oz* directly into *Scala*, by expanding its syntax. The brilliant success of this project did not only open him the doors of the EPFL (École polytechnique fédérale de Lausanne, the institution behind *Scala*), where he is now the executive director of the Scala Center; it also had a direct impact on the development of the *Scala* language itself.

If anything, this work proved how realistic and important our goal is : reflections on syntax design and programming languages in general inspire other programmers, influence their way of working and thinking, and actively impacts the future of programming as a discipline.

1.2.3. NewOz 2020 : the great big jump

Bolstered by the success of the *Ozma* project, the thesis of Jean-Pacifique Mbonyincungu started with the main objective to "create, elaborate and motivate a new syntax" [Mbo20] for *Oz*. It did so 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 thesis³ 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 the others.

²The reader will find some interesting opinions and figures on this polarizing subject in the bibliography, at [Hao20], [Kra16], [sro20], and [BV21]

³See the appendix C.2 of last year's thesis [Mbo20]

- 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 it 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.

1.3. Contributions of this thesis

This work provides main results which can be summarized as follows :

- Make further adaptations to the M. Mbonyingungu's *NewOz 2020* syntax, by addressing points that were left open last year, or by expanding the reflection on other elements;
- Creating a compiler for *NewOz* : even if last year's thesis made work in that direction, we felt like a more robust solution was necessary to gain acceptance around *NewOz*;
- Gathering, for the first time, feedback from the community on the new syntax : it is indeed crucial to leverage the experience and opinions of numerous programmers when designing a syntax, especially from people outside our close social and professional circle;
- Conducting a broad reflection on what is necessary to design a good syntax, why it is important, and how this thesis enters into an ambitious, long-term goal of creating an improved and accepted syntax for the *Oz* programming language.

1.4. Conclusions and the road ahead

Finally, we will conclude this work by reflecting on the quality of our results, in an honest manner taking into account missed opportunities and genuine mistakes, but also time and physical constraints of the project. In a second time, we will then provide elements to help potential future works on this topic, in the form of ideas to explore, projects to take inspiration from, and goals to achieve.

In particular, we hope to demonstrate that placing this thesis in a multi-year process is not only the best to alleviate the intrinsic time limitations posed by the format of master theses, but also how it is simply the best way to carry out such reflections on computer language design in general, and complex, multi-paradigm languages in particular.

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.

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, we feel like *NewOz 2020* has been successful : this new syntax feels modern and more in par with the syntax's of languages 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 extensive code examples covering those

changes in appendix C, 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

As mentioned above, we will not cover the thought process behind every syntactic element in detail, as it would be a repetition of the work done in M. Mbonyincungu's thesis. However, we still feel it is important to briefly describe them, in order to provide a global overview of *NewOz 2021* in a single place.

You will thus find some small code examples targeting a specific syntactic element; the interested reader can find complete program examples in appendix C.

In this section, code examples will present *Scala* and *Ozma* as a single entity, since, as we presented before, *Ozma* is an extension of *Scala*'s syntax; on the other hand, *NewOz 2020* and *2021* snippets, when applicable, will be separated to emphasize their differences and highlight the specific contributions made in this year's version.

2.2.1. Variables and values

A first syntax element we reviewed in *NewOz 2020* was the declaration and use of variables and values. While the introduction 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 to us precarious at best; not only does it go against 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.

Another idea that was left out in *NewOz 2021* was the support for variables in both uppercase and lowercase; this idea seems problematic to us because it goes against the conventions used by most programming languages. Not keeping this in the new syntax also allows us to save capitalized nouns for class names (see later).

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 `vars`, 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

Oz	Scala/Ozma
<pre> local X Y = 8 Z = {NewCell 5} in Z := Y end </pre>	<pre> ... { val x: Int val y = 8 var z: Int = 5 z = y } </pre>
NewOz 2020	NewOz 2021
<pre> ... { val x; val Y = 8 var z = 5 z := Y } </pre>	<pre> ... { val x val y = 8 var z = 5 z = y } </pre>

Figure 2.1.: Variables and values syntax comparison

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 **vars** and **vals** 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 **vars** 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. A comparison of the successive versions of this syntax can be found in Figure 2.1.

2.2.2. Functions and procedures definition

In *Oz*, functions are in fact a subclass of procedures. This allows for a lot of flexibility in the way they are called, since it allows for the ability to store the result in a variable, or to return it directly. Consequently, it allows for the call to be a statement or an expression, depending on what the situation demands.

However, there is still a syntactic difference between the two : the keywords used to declare them highlights this contrast, and provide an important educational, but also visual, value. The decision was thus made by M. Mbonyincungu to keep two separate keywords for the two use cases, albeit not the same : "fun" becomes "def" and "proc" becomes "defproc".

More importantly, the way those functions and procedures are called has been completely overhauled, to align with *Scala* and other modern languages. In particular, functions are now called using their (non-capitalized) name, followed by the comma-separated arguments list enclosed in parentheses. Similarly, their definition also requires the argu-

Oz	Scala/Ozma
<pre> local F P in proc {P X Y} Y = 2*X end fun {F X} 2*X end {Browse {F 1}} end </pre>	<pre> ... { def p(x:Int, y:Int):Unit => {y = 2*x} def f(x:Int):Int => {return 2*x} println(f(1)) } </pre>
NewOz 2020/2021	
<pre> ... { defproc p(x, y) {y = 2*x} def f(x) {2*x} browse(f(1)) } </pre>	

Figure 2.2.: Functions and procedures syntax comparison

ments to be separated by commas. This feels way better than the old curly braces-based method, on top of improving consistency with the objects syntax (see later). A comparison of the successive versions of this syntax can be found in Figure 2.2.

2.2.3. Data structures

The syntactic elements related to data structures like lists, records, and tuples haven't seen many changes compared to *NewOz 2020*. Lists can now be defined using a square brackets-enclosed, comma-separated list of values, in a desire to align with the way functions and procedures arguments are now defined (see before).

Numeric labels (implicit or not) for records, tuples, and trees are fully supported in *NewOz*, which is something that wasn't possible in *Scala* but existed in *Oz*. Finally, those labels must begin in *NewOz* with an apostrophe, in order to distinguish them from method calls.

A comparison of the successive versions of this syntax can be found in Figure 2.3.

2.2.4. Mathematical elements

All mathematical operations in *NewOz* are directly inherited from *Scala*. In particular, the keywords "orelse" and "andthen" in *Oz* have been replaced by the more modern "||" and "&&", since the confusion with the pipe symbol "|" has been lifted following the syntactic changes made to lists (see above). Some other weird particularities of *Oz* were corrected, like the comparisons operators "<=" now being replaced by "<=", or the minus prefix-operator finally using the minus "-" sign instead of a tilde "~".

A comparison of the successive versions of this syntax can be found in Figure 2.4.

Oz	Scala/Ozma
<pre> local L1 L2 L3 R T A in L1 = 3 5 1 nil L2 = [3 5 1] L3 = L1 L2 % Numeric labels are implicit R = l(a:10 10) A = l.1 T = 1#2#3 end </pre>	<pre> ... { val l1 = 3::5::1::Nil val l2 = List(3,5,1,Nil) val l3 = l1 :: l2 // Numeric labels are not supported case class L(a:Int, b:Int) var c = L(10,10) val a = c.b val t = (1,2,3) } </pre>

NewOz 2020/2021
<pre> ... { val l1 = 3::5::1::nil val l2 = [3,5,1] val l3 = l1 :: l2 // Numeric labels are supported var c = 'l('a:10, 10) val a = c.1 val t = (1#2#3) } </pre>

Figure 2.3.: Data structures syntax comparison

2.2.5. Lambdas

Another element that underwent heavy changes was the way *NewOz 2020* handled lambda functions and procedures. As M. Mbonyingungu 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 C, *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 keywords ”fun” and ”proc”, or rather their replacement ”def” and ”defproc”, would be preferable.

We also think that this ”arguments => body” construction, while it fits very 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

Oz	Scala/Ozma
<pre> local A=0.0 B C D Maximum in C = {Sin A} D = ~1.2 B = (D<=C) if (({IsFloat D} andthen {IsFloat C}) or else {Not B}) then Maximum = {Max D C} end end </pre>	<pre> ... { val a=0.0; val b:Any; val d:Any; val maximum:Any val c = Math.sin(a) d = -1.2 b = (d<=c) if (d isInstanceOf Float && c isInstanceOf Float) !b) { maximum = Math.max(d, c) } } </pre>
NewOz 2020	NewOz 2021
<pre> ... { //the library methods used weren't actually available in NewOz 2020 var a=0.0, b, d, maximum val c = sin(a) d = ~1.2 b = (d<=c) if ((isFloat(d) && isFloat(c)) isInt(a)) { maximum = max(d, c) } } </pre>	<pre> ... { //the library methods used here are available in NewOz 2021 var a=0.0, b, d, maximum val c = sin(a) d = -1.2 b = (d<=c) if ((isFloat(d) && isFloat(c)) isInt(a)) { maximum = max(d, c) } } </pre>

Figure 2.4.: Basic operations syntax comparison

the new functions/procedures definition, and the revamped code blocks, represent. A comparison of the successive versions of this syntax can be found in Figure 2.5.

2.2.6. Object-oriented features

The syntax elements linked to object-oriented programming haven't seen many changes. The way of 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, as class attributes are in fact a syntactic sugar for cells. 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 it is necessary. This improvement was actually discussed by M. Mbonyingungu 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* (using the apos-

Oz	Scala/Ozma
<pre> local Ex1 in Ex1 = proc {\$ X Y} X*Y end {Browse {Ex1 6 4}} end </pre>	<pre> ... { val ex2 = (x:Int, y:Int) => x * y println(ex2(6, 4)) } </pre>
NewOz 2020	NewOz 2021
<pre> ... { val ex2 = (x, y) => {x * y} browse(ex2(6, 4)) } </pre>	<pre> ... { val ex2 = defproc \$ (x, y) {x * y} browse(ex2(6, 4)) } </pre>

Figure 2.5.: Lambdas syntax comparison

trophe ””) 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).

In *NewOz 2021*, class methods must now use the keyword ”defproc”. *NewOz 2020* used ”def”, which is misleading in our opinion since class methods cannot return values in *Oz*; they are procedures, and we feel like this syntactic change is also a better way to explicit this than the ”meth” keyword used in *Oz*.

Finally, object application now longer uses the comma ”,”, symbol, but is now expressed using a dot ”.” instead, which feels way more natural to anyone that has used other languages with object-oriented capabilities.

A comparison of the successive versions of this syntax can be found in Figure 2.6.

2.2.7. Conditions and pattern-matching

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 makes the code easier to read.

Similarly, the **catch** clauses 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. A comparison of the successive versions of this syntax can be found in Figure 2.7.

Oz	Scala/Ozma
<pre> class Counter attr value pm:PrivateMethod % The private method is now accessible % to children through the attribute meth inc(I) value := @value + I end meth PrivateMethod(X) {Browse X} end meth incr {self inc(1)} end end class Child from Counter Other meth superCall Counter,inc(5) end end </pre>	<pre> class Counter { var value:Int val pm:Unit = privateMethod def inc(i) = { value = value + i } private def privateMethod(val x:Int)={ println(x) } def incr() = { this.inc(1) } } class Child extends Counter with Other { def superCall() = { super.inc(5) } } </pre>
NewOz 2020	NewOz 2021
<pre> class Counter { attr value; attr pm = PrivateMethod def 'inc(i) { value := @value + i } def PrivateMethod(x) { browse(x) } def incr() { this.inc(1) } } class Child extends Counter, Other { def superCall() { super(Counter).inc(5) } } </pre>	<pre> class Counter { attr value attr pm = PrivateMethod def inc(i) { value = value + i } def PrivateMethod(x) { browse(x) } def incr() { this.inc(1) } } class Child extends Counter, Other { def superCall() { super(Counter).inc(5) //In case of single inheritance : super.inc(5) } } </pre>

Figure 2.6.: Classes and objects syntax comparison

Oz	Scala/Ozma
<pre> local L=[1 3 5] in if {Contains L 3} then {Browse 'Has 3'} elseif {Contains L 5} then {Browse 'Has 5 but no 3'} end % case L of 1 L2 then {Browse 'Case 1'} [] 3 L2 then {Browse 'Case 2'} else {Browse 'Default case'} end end </pre>	<pre> val l = List(1, 3, 5) if (l.contains(3)) {println("Has 3")} else if (l.contains(5)) { println("Has 5 but not 3") } // This code is not quite equivalent because Scala lists aren't recursive structures : l(1) match { case 1 => println("Case 1") case 3 => {println("Case 2")} case _ => println("Default case") } </pre>
NewOz 2020	NewOz 2021
<pre> var l = [1, 3, 5] if (contains(l, 3)) {browse("Has 3")} else if (contains(l, 5)) { browse("Has 5") } // match l { case 1::l2 => browse("Case 1") case 3::l2 => {browse("Case 2")} else browse("Default case") } </pre>	<pre> var l = [1, 3, 5] if (contains(l, 3)) {browse("Has 3")} else if (contains(l, 5)) { browse("Has 5") } // match l { case 1::l2 => {browse("Case 1")} case 3::l2 => {browse("Case 2")} else {browse("Default case")} } </pre>

Figure 2.7.: Conditional structures syntax comparison

Oz	Ozma
<pre> local A B E in thread {Compute A @B} end lock A then C in C=@B B:=C+1 end raise E end end </pre>	<pre> ... { thread { compute(a, b) } // Locks do not have nice such a nice syntactic support in Scala throw e; } </pre>

NewOz 2020/2021

```

... {
  thread { compute(a, b) }
  lock (a) {
    var c
    c = b
    b = c+1
  }
  raise {e}
}

```

Figure 2.8.: Threads, locks and exception raising syntax comparison

2.2.8. Other structures

Other syntactic structures have undergone some changes, whose motivation was simply to make them match with what had been modified before. In particular, the syntax for threads, locks, and the "raise" statement was slightly adapted to incorporate the new scope system using curly braces.

A comparison of the successive versions of this syntax can be found in Figure 2.8.

2.2.9. Built-in library

Another important improvement was the import of the complete *Oz* standard library, as described in the official Mozart documentation¹, into the compiler itself. The Parser from M. Mbonyingungu only supported a subset of pre-defined functions that were put in 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.

¹Mozart'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]

2.3. In the end : a self-evaluation

[TODO] Briefly, what we think of our syntax at this stage

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).

Figure 3.1.: Schematic representation of a classic compiler model

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, We feel like 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; We 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 last 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 library¹, 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

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

²See sections 3.2.3 and 3.3.1 of [Mbo20]

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 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. Mbonyingungu 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 $(p1 \mid p2)$, 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 *noz*, 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. Mbonyingungu'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 [Van21b], 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

TODO - reusing the same image as above, but slightly modified to nozc

Figure 3.2.: Schematic representation of the structure of the `nozc` compiler

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 guidelines³. The user also has the ability to see the intermediary *Oz* code generated during the compilation, or even to personalize the logging level of the output, by using the well-known Apache's Log4j logging levels⁴.

The interested reader will find in appendix D a small example of the compilation process in `nozc`.

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

³More information on those practices can be found at <https://clig.dev/#philosophy> [Pa21]

⁴To 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]

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 conventions⁵. 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 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 JRE⁶. 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.
- Gradle⁷, 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 Rules⁸ was used for some specific test cases.

Overall, a great emphasis has been put on making `noz` 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,

⁵The online documentation for Picocli is located at <https://picocli.info/> [Pop21]

⁶An overview of JavaCC’s features can be found at <https://javacc.github.io/javacc/> [VS21]

⁷Gradle’s homepage is located at <https://gradle.org/> [Inc21b]

⁸This 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]

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 license⁹.

3.6. Evaluation of our approach

We are convinced that the approach we selected with `noz` makes it a great tool for the future contributors who will continue to work on *NewOz*'s syntax in the coming 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 and/or column number to the programmer. However, this problem will progressively disappear over time with the maturation of `noz`, 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 `noz` output JVM bytecode, effectively removing the need for "manual" multi-platform support. However, this approach would also come with its own drawbacks and

⁹This license is available for consultation at <https://github.com/MaVdbussche/nozc/blob/master/LICENSE>

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 are a lot of gains to be made here, especially from a teaching perspective. This would probably be a massive undertaking 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 still 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.

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

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 approach we followed to gather feedback from various developers from both in and outside our network, we will then give a first critical evaluation of this process 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

Before describing our evaluation approach, it is important to describe what its objectives were, and what a perfect evaluation would have looked like.

One of the main goals of this thesis, as we briefly mentioned in the introduction chapter, was to gather, for the first time, feedback from people unfamiliar to the project. Specifically, we wanted to collect opinions on the syntax as it stands at this point in time, after two successive years of work on it. The importance of this process can't be overstated as, like in other matters, an outsider's opinion often brings a new perspective on things, pointing a finger on what seemed like an unimportant detail, and asking uncomfortable questions that forces us to reevaluate our stance.

A syntax can't be designed lightly : if it is to stand the test of time, it should be conceived organically, by gathering feedback and adjusting specific elements, in an iterative process that can (and should) take a long time. This is the best way to obtain a result that satisfies as many people as possible; in turn, this means it will be used by a lot of programmers *because it suits their needs*. After all, we have to remember that programming languages exist to solve real issues people face, be it in a professional or an educational environment; this is not a purely theoretical exercise designed by some computer scientists to challenge themselves.

In the light of this, our intention was to put together opinions from as many programmers as possible, and we first took the time to carefully design our evaluation process. Two main issues now had to be tackled : (a) contacting those people and sparking their interest in the project; (b) finding an effective way to gather their opinions, while allowing a real debate to take place between contributors.

The first point was fairly easy to address, and we send messages through different channels : mailing lists of EPL alumni, private messages to friends working in STEM, as well as Professor Van Roy’s *Twitter* account, on which the message reached a couple of hundreds of people.

The second one demanded a bit more work. We decided to use the *issues* feature of *GitHub* to host the discussions, for multiple reasons :

1. It is a website that tech-savvy people generally trust and know how to use, at least on a basic level;
2. It is a highly customizable platform, where issues can be categorized with labels, linked with each other, or cited from elsewhere;
3. *Github* is available in all countries, and has taken specific actions to limit the likelihood of it being blocked in certain parts of the world¹;
4. Most potential contributors will already have a *GitHub* account; if not, creating one is free and easy to do.

It made sense to host this discussion on the *GitHub* repository already hosting the code for the *noz* compiler. However, we firstly had to create an extensive documentation around the language, with tutorials and code examples, to help contributors get started with *NewOz*. This documentation is also available on the same repository, at [Van21b].

4.1.1. Results of this approach

In this section, we will provide a rapid summary of contributions we received from the community; the interested reader can consult (and participate in !) the full discussion online. All of these opinions and ideas were gathered on *GitHub*, as we said, and will remain available at [Van21b]. The contributors keep full intellectual credit for their contribution; we simply compile them here in a succinct manner for the purposes of the discussion.

A first suggestion that was brought to the table was the addition of a “**return**” keyword. It was mentioned how, being only allowed in functions, it would allow the user to keep in mind the differences with procedures. It could also make the creation of future syntax highlighting tools easier by clearly identifying the last statement of a function. However, we identified two problems with this suggestion :

- Function bodies are expressions in *Oz*; this change would not fit nicely in that perspective, making the body of all methods essentially a statement;

¹Readers interested in this topic can consult the repository at <https://github.com/github/gov-takedowns> for an example of such actions

- Distinguishing functions from procedures is arguably easy enough, thanks to the use of separate keywords in their respective definition, but also because of the point above.

The point about future tooling like syntax highlighting was indeed important to mention; even though the language is still far from a state where a proper software ecosystem develops around it, it is still important to keep this sort of things in mind from the start. In this specific case, the absence of a return keyword means that some sort of analysis phase needs to be performed on the program to determine if a phrase is an expression or a statement, which can in turn determine if it is a suitable last phrase in a function. It does not seem to us like a big issue, even though we admittedly don't have much knowledge about this type of tools. Another option could be to copy the behavior of *Scala*, which seems to make the `"return"` keyword optional; but this comes with its own problems².

Another talking point was the way attributes in classes are expressed. In the current version of the syntax, they must be declared before the opening of the class' scope (see the code examples in appendix C), with a repetition of the `"attr"` keyword for each new attribute. This syntax was rightfully deemed redundant, and it was suggested to take inspiration from *C#'s* properties syntax to design a more elegant approach³. More simply, declaring them in a sequence, separating them with commas, could be another option, that would be more akin to the way variables and values are declared.

It was also mentioned that the choice of the keywords `"var"/"val"` might not be most wise, as they look very similar, which might pose problems for some people. Different options were proposed, like `"mut"/"val"`, `"let mut"/"let"`, or even `"cel/cell"/"var"`, in a witty nod to *Oz's* cells feature. A lot of arguments can be made for each of those propositions and others; the good point here is that it does not impact the language's philosophy in any way, nor its implementation significantly, as modifications to the lexical grammar have minimal consequences on the compiler's implementation. Changes to it will, however, break compatibility with existing programs, something that has to be kept in mind in future iterations of the syntax.

Additionally, another good point was made regarding a visually confusing syntax element : the labels and features of records. The current version of the syntax uses an apostrophe `"` in front of the non-capitalized name, which is necessary to make the distinction with variable names or method calls. However, it was brought to our attention how this could be not only be too subtle for people with poor eyesight, but also how it could be confused with strings in general. While we agree that the current solution is not satisfactory, we could not find a suitable replacement yet : most characters on the keyboard are either already used, or are not standard enough to appear on all keyboard

²See in that regard the answer from user dhg on *StackOverflow* (<https://stackoverflow.com/a/12560532>), but also the blog post from Rob Norris at [Nor14]

³See <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties>

layouts. A syntax using chevrons was suggested, but we didn't take it on as it would create a lot of problems with *HTML*-like content; in particular, online documentation and emails displaying code snippets would see their formatting disrupted if these particular characters were used without careful escaping.

Finally, we will quickly mention the problem that was brought to our attention regarding Unicode support in *Mozart*. Even though `noz` could natively support all Unicode characters, the fact that the Mozart2 compiler doesn't, poses a problem : a variable named `"bμ"` can't be compiled by the underlying `ozc` compiler. This problem could be circumvented by "translating" problematic characters, giving something like `"B_C2B5_"` in output of `noz`. We find this solution acceptable since it probably won't enter into play very often; nonetheless, it is an important addition to include in a future version of `noz`.

4.2. First evaluation and adjustments

Our general takeaway on the feedback we received is the following : we didn't get the high-level, philosophical reflections we expected, but the fault probably lies in our ill-suited approach for a debate on those subjects.

In terms of content, we hoped for more content-focused reactions on the general philosophy of the language. 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 their 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. In-depth reflections can only come after extensive use of the syntax, from people having written different programs using various paradigms. In that regard, calling upon the online community to help us in a deep reflection on the philosophy of a syntax was probably a process that was doomed to fail.

Secondly, the "philosophy" behind the *NewOz* maybe wasn't explicit enough in the first place; how then could users react upon it ? The debate would probably have benefited from a deeper high-level presentation of the language in the documentation, similar to the extensive syntax tutorial that was written. Such a document could have presented our vision more explicitly, which would have allowed contributors to give us some of their opinions on it.

With all this being said, the remarks we did gather still 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 even fun programming syntax is.

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

[TODO] Broad reflection on how syntax design should happen. Did we do a good job at that ? We see two ways to do it : (a) start from Oz, "translate" it piece by piece with various inspirations (while keeping consistency) (this is what we did with newOz) (b) Take an existing language, and add what we need on top of it (e.g. Ozma, FlowJava). This ensures consistency, but is it really its own language at this point ? Explicit our opinion on this matter.

- We are but one step in a large-scope work (long-term project/collaboration) - spread over multiple master's projects
- Toward ultimate goal of a new, improved, and accepted syntax for the Oz multiparadigm language (Multiparadigm = now it is accepted that languages must be multiparadigm-Java has lambdas, Scala is functional-objet, Cloud analytics combine functional, concurrent, and database structure)
- Be honest - it is hard to design a syntax - problems encountered, why this is a multi-year project -> what we did to alleviate these challenges
- Final, definitive way of formulating a multiparadigm language. Oz was a pioneer, followed by Scala, Ozma, etc., but what will multiparadigm languages look like in the future? In the future when all languages are multiparadigm. We are making steps toward this - take Oz original ideas but with new syntax inspired by existing languages. Lyric goal : here we made one small step in this (Apollo reference?)

Some items of this list probably have their place in the next chapter !

5. Conclusion

[TODO] **Complete this section** Resume our approach and each chapter briefly 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)

- integrate in a course ? Gather feedback from students working a medium-scale project-> after some weeks/months, they might give more high-level feedback ?
- New version of [VH04] ?
- Integration in Mozart/Emacs
- Other things left to do/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: <https://www.info.ucl.ac.be/~pvr/MemoireSebastienDoeraene.pdf>.
- [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>.
- [Nor14] Rob Norris. *The Point of No Return*. May 9, 2014. URL: <https://tpolecat.github.io/2014/05/09/return.html>.
- [Kra16] Moshe Kranc. *The Rise and Fall of Scala*. Oct. 6, 2016. URL: <https://dzone.com/articles/the-rise-and-fall-of-scala>.
- [Hao20] Li Haoyi. *The Death of Hype: What’s Next for Scala*. Apr. 10, 2020. URL: <https://www.lihaoyi.com/post/TheDeathofHypeWhatsNextforScala.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>.
- [sro20] JetBrains s.r.o. *The State of Developer Ecosystem 2020*. 2020. URL: <https://www.jetbrains.com/lp/devecosystem-2020/>.
- [BV21] TIOBE Software BV. *TIOBE Index for May 2021*. May 2021. URL: <https://www.tiobe.com/tiobe-index/>.
- [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>.
- [Inc21c] Lightbend Inc. *Akka*. 2021. URL: <https://akka.io/docs/>.
- [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>.
- [Wik21c] Wikipedia. *Scala (programming language)*. May 24, 2021. URL: [https://en.wikipedia.org/wiki/Scala_\(programming_language\)](https://en.wikipedia.org/wiki/Scala_(programming_language)).

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. *NewOz: Steps toward a modern syntax for the Oz programming language*. GitHub repository. 2021. URL: <https://github.com/MaVdbussche/Master-Thesis>.
- [Van21b] Martin Vandenbussche. *Nozc*. GitHub repository. 2021. URL: <https://github.com/MaVdbussche/nozc>.

Appendices

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

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

This EBNF grammar is thus suitable for recursive descent, and is the grammar used by *nozc*.

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 ϵ with the set of words w (optional group)
$\{w\}$	zero or more times w
$\{w\}^+$	one or more times w
$w_1 w_2$	concatenation of w_1 with w_2
$w_1 w_2$	logical union of w_1 and w_2 (OR)
$w_1 - w_2$	difference of w_1 and w_2

```
interStatement ::= {nestConStatement}+  
                | DECLARE inStatement EOF
```

```
statement ::= nestConStatement  
            | SKIP
```

```
expression ::= nestConExpression  
            | nestDecAnonym  
            | DOLLAR  
            | term  
            | THIS  
            | inExpression
```

```
parExpression ::= LPAREN expression RPAREN
```

```
//Declarations still need to come first (keeps Oz' idea)
```

```
inStatement ::= LCURLY {declarationPart} {statement} RCURLY
```

```

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

nestConStatement ::= assignmentExpression
    | variable LPAREN [expression {COMMA expression}] RPAREN
    | inStatement
    | 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 ::= IF LPAREN expression RPAREN inExpression
    {ELSE IF LPAREN expression RPAREN inExpression}
    [ELSE inExpression]
    | MATCH expression LCURLY
      {CASE caseExpressionClause}+
      [ELSE inExpression]
    RCURLY
    | TRY inExpression
      [CATCH LCURLY
        {CASE caseExpressionClause}+
      RCURLY]
      [FINALLY inStatement]
    | RAISE inExpression
    | THREAD 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 ::= DEFPROC methHead [ASSIGN variable] (inExpression|inStatement)
  | classDescriptor

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

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

assignmentExpression ::= conditionalExpression

assignmentStatement ::= variable ASSIGN expression

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 ::= (MINUS|PLUS) unaryExpression
                | simpleUnaryExpression

simpleUnaryExpression ::= LNOT unaryExpression
                    | postfixExpression

postfixExpression ::= primary

primary ::= variable | int | float | character | string | UNIT | TRUE | FALSE |
          UNDERSCORE | NIL
          | variable LPAREN [expression {COMMA expression}] RPAREN
          | variable DOT variable [LPAREN [expression {COMMA expression}] RPAREN]
          | THIS DOT variable LPAREN [expression {COMMA expression}] RPAREN
          | SUPER [LPAREN variableStrict RPAREN]
          | DOT variable LPAREN [expression {COMMA expression}] RPAREN
          | parExpression

term ::= assignmentExpression
      | atomLisp LPAREN [[feature COLON]pattern {COMMA [feature COLON]pattern}
                        [COMMA ELLIPSIS]] RPAREN
      | LPAREN expression {HASHTAG expression}+ RPAREN
      | LPAREN expression {COLCOL expression}+ RPAREN
      | LBRACK [expression {COMMA expression}] RBRACK

pattern ::= variable | int | float | character | string | UNIT | TRUE | FALSE |
          UNDERSCORE | NIL
          | atomLisp LPAREN [[feature COLON]pattern {COMMA [feature COLON]pattern}
                        [COMMA ELLIPSIS]] RPAREN
          | LPAREN pattern {HASHTAG pattern}+ RPAREN
          | LPAREN pattern {COLCOL pattern}+ RPAREN
          | LBRACK [pattern {COMMA pattern}] RBRACK
          | LPAREN pattern RPAREN

declarationPart ::= (VAL|VAR) variable [ASSIGN expression]
                  {COMMA variable [ASSIGN expression]}
                  | nestDecVariable

loopDec ::= variable IN expression DOTDOT expression [SEMI expression]
          | variable IN expression SEMI expression [SEMI expression]
          | variable IN expression

feature ::= atomLisp

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

```



```
methHead ::= (variableStrict | variable)
           LPAREN [methArg {COMMA methArg} [COMMA ELLIPSIS]] RPAREN

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

B. Appendix B : Lexical Grammar (2021 version)

This is the lexical grammar used by nozc.

Notation	Meaning
=====	
ϵ	singleton containing the empty word
(w)	grouping of regular expressions
$[w]$	union of ϵ with the set of words w (optional group)
$\{w\}$	zero or more times w
$\{w\}^+$	one or more times w
$w_1 w_2$	concatenation of w_1 with w_2
$w_1 w_2$	logical union of w_1 and w_2 (OR)
$w_1 - w_2$	difference of w_1 and w_2
 <i>// White spaces - ignored</i>	
WHITESPACE ::= (" " "\b" "\t" "\n" "\r" "\f")	
 <i>// Comments - ignored</i>	
("/*" { ~("\n" "\r") } ("\n" "\r" "\r\n")	
<i>// Multi-line comments - ignored</i>	
"/*" { CHAR - "*" } "*/"	
 <i>// Reserved keywords</i>	
AT	::= "at"
ATTR	::= "attr"
BREAK	::= "break"
CASE	::= "case"
CATCH	::= "catch"
CLASS	::= "class"
CONTINUE	::= "continue"
DECLARE	::= "declare"
DEF	::= "def"
DEFPROC	::= "defproc"
DO	::= "do"
ELSE	::= "else"
EXPORT	::= "export"
EXTENDS	::= "extends"
FALSE	::= "false"
FINALLY	::= "finally"
FOR	::= "for"

```

FROM      ::= "from"
FUNCTOR   ::= "functor"
IF         ::= "if"
IMPORT    ::= "import"
IN         ::= "in"
LAZY      ::= "lazy"
LOCK      ::= "lock"
MATCH     ::= "match"
NIL       ::= "nil"
OR        ::= "or"
PROP      ::= "prop"
RAISE     ::= "raise"
RETURN    ::= "return"
SKIP      ::= "skip"
SUPER     ::= "super"
THIS      ::= "this"
THREAD    ::= "thread"
TRUE      ::= "true"
TRY       ::= "try"
UNIT      ::= "unit"
VAL       ::= "val"
VAR       ::= "var"

// Operators
ASSIGN     ::= "="
PLUSASS    ::= "+="
MINUSASS   ::= "-="
EQUAL      ::= "=="
NE         ::= "\\="
LT         ::= "<"
GT         ::= ">"
LE         ::= "<="
GE         ::= ">="
IMPL       ::= "=>"
LAND       ::= "&&"
LOR        ::= "||"
LNOT       ::= "!"
MINUS      ::= "-"
PLUS       ::= "+"
STAR       ::= "*"
SLASH      ::= "/"
MODULO     ::= "%"
HASHTAG    ::= "#"
UNDERSCORE ::= "_"
DOLLAR     ::= "$"
APOSTROPHE ::= "'"

```

```

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

// Literals
VARIABLESTRICT ::= UPPERCASE{ALPHANUM}
                | "\"" (ESC | PSEUDO_CHAR | ~("\"" | "\\" | "\n" | "\r" ) ) "\""
VARIABLE       ::= LOWERCASE{ALPHANUM}
ATOM           ::= (ATOMLISP | "~" (ESC | PSEUDO_CHAR | ~("\"" | "\\" | "\n" | "\r" ) ) "~")
ATOMLISP       ::= "'" {LETTER}
STRING         ::= "\" { ESC | PSEUDO_CHAR | ~("\"" | "\\" | "\n" | "\r" ) } "\""
CHARACTER      ::= (DEGREE(CHARCHAR | PSEUDO_CHAR)
                | "'" (ESC | ~("'" | "\\" | "\n" | "\r" ) ) "'" )
INT            ::= (DECINT | HEXINT | OCTINT | BININT)
FLOAT          ::= {DIGIT}+ DOT {DIGIT} [ ( "e" | "E" ) [ "~" ] {DIGIT}+ ]
UPPERCASE     ::= "A" | ... | "Z"
LOWERCASE     ::= "a" | ... | "z"
LETTER        ::= "A" | ... | "Z" | "a" | ... | "z"
DIGIT         ::= "0" | ... | "9"
NON_ZERO_DIGIT ::= "1" | ... | "9"
CHARINT       ::= ( "0" | ... | "9" ) | ( "1" | ... | "9" ) ( "0" | ... | "9" )
                | "1" ( "0" | ... | "9" ) ( "0" | ... | "9" )
                | "2" ( "0" | ... | "4" ) ( "0" | ... | "9" ) | "25" ( "0" | ... | "5" ) // (0-255)
ALPHANUM      ::= (UPPERCASE | LOWERCASE | DIGIT | UNDERSCORE)
DECINT        ::= ( "0" | (NON_ZERO_DIGIT{DIGIT}) )
HEXINT        ::= "0" ( "x" | "X" ) {HEXDIGIT}+
OCTINT        ::= "0" {OCTDIGIT}+
BININT        ::= "0" ( "b" | "B" ) {BINDIGIT}+
OCTDIGIT      ::= "0" | ... | "7"
HEXDIGIT      ::= (DIGIT | ( "A" | ... | "F" ) | ( "a" | ... | "f" ) )
BINDIGIT      ::= ( "0" | "1" )
ESC           ::= "\\" ESCAPE_CHAR
ESCAPE_CHAR   ::= ( "a" | "b" | "f" | "n" | "r" | "t" | "\\" | "'" | "\"" | DEGREE )
CHARCHAR      ::= ~("\"")

```

```
// In the classes of words <variable>, <atom>, <string>, and <character>, we use
// pseudo-characters, which represent single characters in different notations.
PSEUDO_CHAR ::= ( "\\"(OCTDIGIT)(OCTDIGIT)(OCTDIGIT) ) | (
    "\\(" "x" | "X" )(HEXDIGIT)(HEXDIGIT) )
// End of file
EOF ::= "<end of file>"
```

C. 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
- try..catch..finally

D. Appendix D : Compilation example

E. Appendix E : Documentation and tutorial

Move this as the first appendix ?