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

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. SOURCE : MOZART2.ORG

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, through its *one-fits-all* approach.

However, it has become obvious over time that the syntax of the language 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) that a new syntax would be developed for *Oz*. Reformulate this sentence 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, unique environment. This process has started in 2020, with the master thesis of M. Mbonyincungu[2], who created a first design for the *NewOz* syntax, heavily inspired by Ozma and Scala (see below). This thesis continues this work by making more refinements to the syntax, as well as creating a fully fletched compiler around it.

In the following sections, we will provide an overview of what has been achieved in previous works covering *Oz*'s grammar in general, and *NewOz* in particular. We will describe how those served as a starting point for our reflexions, and the problems or limitations we identified.

DESCRIBE THE TIMELINE OVER THE YEARS + FUTURE Describe process of community feedback gathering What were the ambitions at the start ?

1.2 Inspirations

Scala - Ozma - NewOz2020 Last year's work of Jean-Pacifique Mbonyincungu had as main objective to "create, elaborate and motivate a new syntax"[2] for *Oz*, by systematically reviewing a lot of 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 reflexion 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 provided two main results :

- The definition of a new syntax (which we will call *NewOz* 2020 in this document), as we said before; this syntax has been described[1] as an EBNF grammar.
- The writing of a Parser, which is able to convert code written in *NewOz* to the equivalent *Oz* code. This Parser is actually a compiler, in the sense that it "translates computer code written in one programming language into another language"[2]. However, it does not have the advantages and flexibility of most compilers, as we will explain in section 3.2.

1.2.1 Current state of the *NewOz* grammar (DELETE)

Talk about pros and cons of Jean-Pacifique's *NewOz*

1.2.2 The *NewOz* Parser

As M. Mbonyincungu explains in his thesis [2], 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 "parser". M. Mbonyincungu decided to go with the second approach, while we selected a third approach that could be described as a mix of both.

DESCRIBE PROS AND CONS OF COMPILER AND PARSER (JPM thesis section 3.2) But this presents multiple difficulties : (1) it is technically more difficult and would take more time than the Talk about pros and cons of Jean-Pacifique's Scala Parser

[1] See the appendix C.2 of last year's thesis [2]
[2] Wikipedia Compiler page ref

1.2.3 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

1.2.4 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

Rappel : ceci est un chapitre d'intro

- Adaptation to JP's syntax
- *NewOz* compiler
- Community feedback

1.4 Conclusions on the new syntax

Rappel : ceci est un chapitre d'intro

2 Design principles of the new syntax

- What is the motivation behind the thesis ? (covered before)
- Why is it important ? (idem)
- What are the ambitions ? (should be covered before)
- What is the general philosophy ? Why yet anther language ? Why not just use Scala at this point ?

MES CONTRIBUTIONS - TRAVAIL TH2ORIQUE Principes derrière la syntaxe + en pratique, review de toutes les features du langage.

3 The *NewOz* Compiler : `nozc`

3.1 A quick introduction to compilers

MAKE THIS section 3.0 How do compilers work in general ? Try to keep it not TOO technical and long. Lexer - Parser/syntax analysis - Semantic analysis - (optimizer) - code generator

3.2 The intial situation (find a better title)

Extract from M. Mbonyincungu's thesis : "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."[2]

Instead, last year's thesis brought forward (???) the idea of writing a syntax parser, 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 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 [...]".[1] 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)."[2] In my opinion, the program created by M. Mbonyincungu doesn't match any of those two definitions [de manière satisfaisante], as we will discuss later; I think it lies somewhere in between those two definitions. But to stay consistent with the vocabulary used in lest year's thesis and avoid confusion, we will refer to it as "the Parser" in the rest of this document.

M. Mbonyincungu's Parser makes use of Scala's Parsing Combinators library[3], which provides a syntax to match regular expressions and describe the relationship between them. The matched expressions are then translated to *Oz* code, with a great emphasis being put on maintaining the code's visual format.[4] This is important because the

[1] Cite correctly : `https://en.wikipedia.org/wiki/Parsing`, consulted on 13/05-11:54
[2] Cite correctly : `https://en.wikipedia.org/wiki/Compiler`, consulted on 13/05-12:06
[3] cite correctly : `https://www.scala-lang.org/api/2.12.3/scala-parser-combinators/scala/util/parsing/combinator/Parsers.html`
[4] See sections 3.2.3 and 3.3.1 of [2]

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 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 its 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 when it comes to variable types, making it impossible, for example, to evaluate if an arithmetic operation is valid for two given arguments.

3.3 The need for something else

"We will now discuss the points 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 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 [fonctionnement]. 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, because the Parser has limited semantic analysis capabilities, most errors are caught during this second phase. This means that the user receives 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 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 Nozc dissected

The *NewOz* Compiler[3], 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. It runs on Windows, MacOS, and Linux, through a command-line interface. The overall approach used by `nozc` is 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. 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. 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. 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[5]. 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[6].

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, future 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[7]. 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 JRE[8]. This

[5] See `https://clig.dev/#philosophy` for an interesting read on the subject
[6] To be exact, `nozc` does not use Log4j itself, 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 description of those levels.
[7] Cite correctly : `https://picocli.info/`
[8] cite : `https://javacc.github.io/javacc/`

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, 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[9] 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.
The program itself is published on GitHub under the BSD license[10].

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 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. In particular, our approach does not free itself from the dependency on the legacy `ozc` compiler, which was one of our criticism towards M. Mbonyincungu's approach. A more mature compiler should be able to generate machine code directly, or at the very

[9] This collection of JUnit 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`

[10] This license is available for consultation at `https://github.com/MaVdbussche/nozc/blob/master/LICENSE`

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 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)[11].
Another solution would be to fork the existing `ozc` compiler and modify its front-end to accept the new syntax.

3.7 What's next for `nozc` ?

Specific areas of the compiler that need improvements:

- Better functors support
- Pretty printing
- See other open issues on GitHub
- Interactive, Mozart/Emacs-style compilation (will be difficult without complete rewrite imho)

Conclusion du chapitre : 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 major first step in that direction.

[11] Further relfexions on this approach might benefit from reading the work of Sébastien Doeraene on Ozma[1]

4 The feedback from the community

This chapter = general community feedback - résumé des suggestions Then we will describe what will/should change for the next version of *NewOz*

Conclusions que nous tirons après 1 mois de ce feedback de la communauté Suggestions pour future work on top of this

========== 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) Did we integrate their feedback ? If so, show intermediary steps until final version If not, why ?

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 ?

Donner une syntaxe finale qui comprend le feedback de la communauté

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)

Appendices

Appendix A : *NewOz* EBNF Grammar

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 this

parExpression ::= LPAREN expression RPAREN

inStatement ::= LCURLY {declarationPart} {statement} RCURLY //TODO added possibility for n
             | 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 bcs of
            | 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 statem

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 remove cha

//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] //TODO no

methHead ::= ([LNOT] variableStrict | atomLisp | UNIT | TRUE | FALSE) //TODO not implement
                [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 that it i

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     ::= "ť" 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*

Appendix D : Documentation and tutorial

Move this as the first appendix ?

Bibliography

[1] Sébastien Doeraene. *Ozma: Extending Scala with Oz Concurrency*. 2003???. Prom. Peter Van Roy. URL: ???.

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

[3] Martin Vandenbussche. *Nozc*. https://github.com/MaVdbussche/nozc. 2021.