

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

Contents

1	Goal of the project and previous works	3
1.1	NewOz (2020 version)	3
1.1.1	Current state of the NewOz grammar	3
1.1.2	The NewOz Parser	3
1.2	Ozma : a Scala extension	4
1.3	Other works	4
2	Philosophy of the new syntax	5
3	The NewOz Compiler : nozc	6
3.1	The need for something else	6
3.2	A quick introduction to compilers	6
3.3	Nozc dissected	6
3.4	Technologies used	7
3.5	Advantages of this approach	7
3.6	Limitations of this approach	7
4	New adaptations to NewOz	8
5	Conclusion	9
	Appendices	10

Abstract *The Oz programming language has proven over the years its value as a learning and research tool **about** programming paradigms, in many universities around the world. It has had a major influence over different 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 Mbonyingu, is to design a brand new syntax for Oz, that will allow the language to tackle new paradigms, while still being compatible with the existing Mozart system.*

1 Goal of the project and previous works

- Describe the project, and the motivation behind the new grammar for Oz. (Use Nozc's README's background section as starting point).

- Previous works done in this context

In this chapter, 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.

- NewOz (Jean-Pacifique's thesis)
- Ozma
- Other previous works on Oz original grammar (used to get the philosophy behind it)

1.1 NewOz (2020 version)

Last year's work of Jean-Pacifique Mbonyincungu used a very systematic approach consisting of 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 discussion that followed, comparing pros and cons of both syntax's, 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.

The two main results of this work were the definition of a new syntax, described in 5 as an EBNF grammar, as well as the writing of a Parser, which is able to convert any code written in NewOz to the equivalent Oz code.

1.1.1 Current state of the NewOz grammar

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

1.1.2 The NewOz Parser

Talk about pros and cons of Jean-Pacifique's Scala Parser

1.2 Ozma : a Scala extension

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

1.3 Other works

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

2 Philosophy 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 another language ? Why not just use Scala at this point ?

Maybe this chapter should be placed **before** the previous one, or as the beginning of it. It feels weird to describe the goal and what has been done, and **then** the philosophy behind said work.

3 The NewOz Compiler : nozc

Extract from Jean-Pacifique'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. Instead, the previous thesis brought forward the idea of writing a syntax parser, that would serve as a compatibility layer between the so-called "newOz" syntax, and the existing Oz syntax.[1] NewOz code will be translated, and then fed to the existing Oz compiler.

This approach has been selected because it is easier to implement, relies on more modern technologies than the existing Mozart compiler, and will allow for more flexibility down the line because of the nature of the underlying Scala code of the Parser. We should however keep in mind the major limitation that this approach brings : error messages generated by the Mozart compiler will be way harder to interpret by the end-user. One of the goals of the project being that this Parser-to-compiler behavior stays transparent to the programmer, we should keep in mind that future Oz learners will know nothing of the current Oz syntax. As such, the compiler output will probably be obscure to them, and it is probably a good idea to try to alleviate this confusion as much as possible.

3.1 The need for something else

Why did we feel like we needed to make a full compiler, and why we didn't go all the way to the machine language

3.2 A quick introduction to compilers

How do compilers work in general ? Try to keep it not TOO technical and long.

3.3 Nozc dissected

General description of the inner workings of the compiler. Do not go in ridiculous details, as the code is well documented and available. Use an example and show its evolution when going through the compiler.

3.4 Technologies used

Why Java (pros and cons) (mention Java here for 1st time -> forces me to be generic in previous section). picocli, JavaCC.

Try to make it short

3.5 Advantages of this approach

Persuade the reader that it is a great platform to reuse and build upon. How useful it can be for future works on the syntax

3.6 Limitations of this approach

Let's keep it real, it is not perfect either...

4 New adaptations to NewOz

Describe changes we made 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
- **Feedback from other users about this.** What did others think of it ? (probably \neq 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 ?

5 Conclusion

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 Mbonyingungu'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 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
 // 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 expression}} 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 new0z
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    ::= "¸" ok
HAT        ::= "^" ok
BOX        ::= "[]" ok
//TILDE    ::= "~" ok
DEGREE     ::= "°" ok
//COMMERCAT ::= "@" ok
//LARROW    ::= "<-" ok
//RARROW    ::= "->" ok
//FDASSIGN  ::= "=: " //skipped
//FDNE      ::= "\=: " //skipped
//FDLT      ::= "<:" //skipped
//FDLE      ::= "=<:" //skipped
//FDGT      ::= ">:" //skipped
//FDGE      ::= ">=: " //skipped
COLCOL     ::= "::" ok
//COLCOLCOL ::= ":::" ok

```


Bibliography

- [1] Jean-Pacifique Mbonyincungu. “A new syntax for Oz”. MA thesis. École Polytechnique de Louvain, UCLouvain, 2020. Prom. Peter Van Roy. URL: <http://hdl.handle.net/2078.1/thesis:25311>.