"A new syntax for Oz"

Mbonyincungu, Jean-Pacifique

ABSTRACT

The premise of this thesis is to "create,elaborate and motivate a new syntax for an education-oriented programming system named "Oz"". So from the conception/researches, passing by multiple debates and analysis over various programming languages, to the implemention/realisation and usage of the new syntax, nothing will shall left aside.

CITE THIS VERSION

Mbonyincungu, Jean-Pacifique. *A new syntax for Oz.* Ecole polytechnique de Louvain, Université catholique de Louvain, 2020. Prom. : Van Roy, Peter ; Mens, Kim ; Laurent, Nicolas. http://hdl.handle.net/2078.1/thesis:25311

École polytechnique de Louvain

A new syntax for Oz

Author: **Jean-Pacifique Mbonyincungu**
Supervisor: **Peter Van Roy**
Readers: **Kim Mens, Nicolas Laurent**
Academic year 2019–2020
Master [120] in Computer Science

Acknowledgements

As you will soon be made aware of, this thesis was made possible thanks to a continuous and ever changing debate on Oz and many programming languages, well known or not. Which is why i would like to sincerely thanks **M.Nicolas Laurent** and **M.Peter Van Roy** who were harsh/fierce but patient debater coupled with great and knowledgeable mentors. Thanks again!

Last but not least, i would like to thanks **M.Kim Mens** whom accepted to be member of the jury of this thesis even though he i contacted him, who was already overbooked, at very close to the dead line. Not all heroes wear capes.

Additional recognition to the various authors of the references cited, I'll happily stand on your shoulders.

Table of Contents

Chapter 1

A new Syntax for Oz

The complete premise of this thesis is to "**create,elaborate and motivate a new syntax for an education-oriented programming system named "Oz"**".

Therefore, this chapter shall be used as an introduction, of sort, meant to familiarise the reader with the ecosystem of the current Oz and the foundations of its new syntax; which we shall name "newOz"[1].

Here, the phrasing "**ecosystem**" is much more fitting than it seems to characterize a programming language; for instance, like in an **ecosystem**:

- **We have a food chain**: some languages or specific syntax are more popular than the others and can force a (non-)explicit trend whether we like them or not.For instance, the really popular $equal$ operator(e.g : `int x = 0; x = x + 2`) which is kind of mathematically paradoxical since the unknown x and can't be itself and superior to itself at the same time. This kind of issue shall be touched upon later but it conveys well the idea that "**a programming language syntax should be scrutinized mostly through the lens of programming languages specter**"
- **There is an overall balance and harmony to respect** : within a given syntax you can't just do anything you want. Keeping as much cohesion as possible is a must (many examples in the following chapter). The main idea here is that, in a more subtle way, there is a difference between a good looking forest and an cluster of good looking trees,phone and flora. **We have thus to be careful that while each piece can look good on their own, the overall painting might not**.

---

[1] Named this way with a lower case at the start for reasons you will be able to guess soon.

- **Not everything can be easily explained at first glance** : often, you have decisions or choices in, for instance, older languages which were made due specific historical reasons("A History of the Oz Multiparadigm Language" n.d.).Such reasons are not that easy to track down nor understand.So when using other programming languages as references, we have to be cautious of such problematic.

The points and comparisons above shall be used as an overture to the overall philosophical and technical side of designing a programming language syntax.

1.1 What is Oz ?

Oz is a multi-paradigm language("A History of the Oz Multiparadigm Language" n.d.) that is designed for advanced, concurrent, networked, soft real-time, or reactive applications.
Oz covers most, if not all, features of many programming paradigms (Seif Haridi, 2013; "A History of the Oz Multiparadigm Language" n.d.):

- **Object-oriented** : including state, abstract data types, objects, classes, and inheritance.
- **Functional programming** : compositional syntax, first-class procedures/functions, and lexical scoping.
- **Logic and constraint programming** : including logic variables, constraints, disjunction constructs, and programmable search mechanisms.
- **Concurrent programming** : for instance , it allows users to dynamically create any number of sequential threads. The threads are dataflow threads in the sense that a thread executing an operation will suspend until all operands needed have a well-defined value.

1.1.1 Learning with Oz

The programming language Oz comes alongside a variety of learning books (like for example, a beefy book named "(*Concepts, Techniques, and Models of Computer Programming* June, 2013)") and are mainly meant to be used as training and teaching material for computer science university students across the globe.

One of the strong point of Oz, from a learning view point, is that the student can learn and get familiar with most programming languages paradigm (e.g : High Order programming, Concurrency, Functional programming, Dataflow, Non determinism , Atomicity, Declarative Programming and so on..) under an efficient programming language (e.g : creating many threads is quite fast and cheap).

While one of the main weakness of Oz is its quite "unique" syntax which is also drawback when learning computer science with the help of Oz.
To give an example, a student could learn a specific programming paradigm but also find him/herself in an not optimal situation due to two reasons (in respect to the thesis):

1. He or she could have a hard time doing the parallel between Oz and the main programming languages currently taught in the early grades of University due to their large syntax discrepancy with Oz(which are, to a name a few, Java , Python, C, C++).
   Such discrepancy can slow down the practical usage of things learned during the course.
2. He or she could lose time learning and getting familiar with a syntax that is quite different from those main programming languages currently used in the schools. Such is could and could not be problematic , it depends on whether learning to use a new syntax from scratch (and specifically , the one of Oz) is a skill more valued than learning the rest faster.

While these issues might not be as that dramatic, the devil often still lies in the details and fixing them can only improve the overall experience.

1.1.2 Philosophies and Concepts behind Oz

Syntactical philosophies and concepts will be a common thread of this thesis[2] due to the fact they are intrinsically linked the overall homogeneity and consistency of a language (or at least a new one, since the natural evolution of an old one can break away from its original purpose for many reasons).

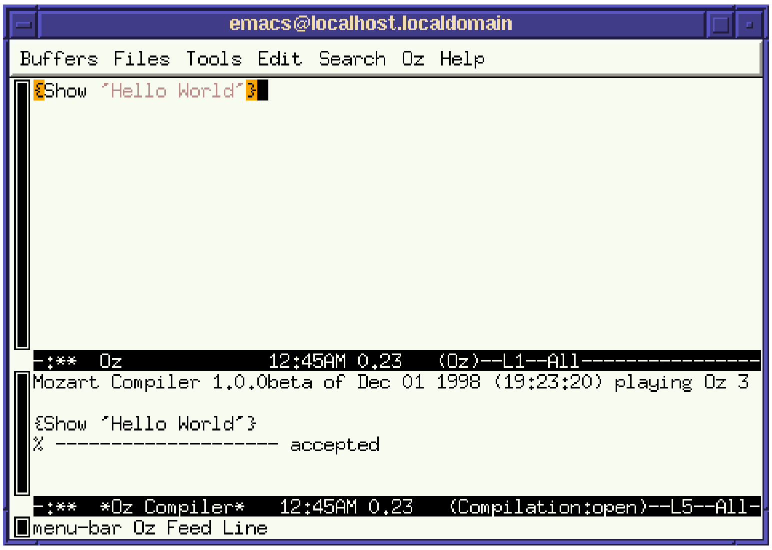
It is thus, when creating a new syntax, important to be aware of the philosophies and concepts of the original Oz to try to keep as many as possible but also to use them adequately when a choice needs to be taken by titling the scale one way or another.

Those ideas are the following (refs : (*Concepts, Techniques, and Models of Computer Programming* June, 2013; Seif Haridi, 2013; “A History of the Oz Multiparadigm Language” n.d.)):

- Oz is an interactive language, which means that like Python or Scala the code can be executed via a live interpreter development Environment (namely **GNU EMACS** for Oz) with a quite distinctive visibility on what is going on behind the code :

---

[2] If you see a **sentence in bold**, it probably means that it is an implicit/explicit philosophy/concept of newOz.



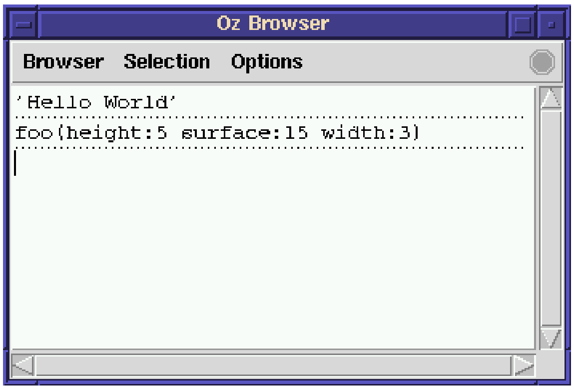
```
{Show 'Hello World'}
-:** Oz              12:45AM 0.23    (Oz)--L1--All----------------
Mozart Compiler 1.0.0beta of Dec 01 1998 (19:23:20) playing Oz 3

{Show 'Hello World'}
% -------------------- accepted

-:** *Oz Compiler*    12:45AM 0.23    (Compilation:open)--L5--All-
menu-bar Oz Feed Line
```

Figure 1.1: Emacs interactive : accepting a statement

And the Mozart system's interactive development environment has also a graphical advantage:



```
'Hello World'
foo(height:5 surface:15 width:3)
```

Figure 1.2: Mozart interactive : showing a variable (can see other data structures too)

Such interactivity is a non negligible advantage when learning programming with the help of Oz.

- The syntax of Oz is meant to be non ambiguous (just from a glance you are able to know what that code is talking about) and straight-forward (tokens don't have multiple meanings and there aren't tens of ways to write the same thing).

Some readers might also ask if keeping these original ideas are still relevant but also how much they should weight on that proverbial scale. The answers to those questions, while they could be taken as arbitrary , are the following :

- Behind the new Oz syntax will still be hidden the old one (at the very least from a compiler viewpoint). It would be counter-productive to voluntarily go against (or even ignore) that fact since it waste the overall potential of the compiler behind.

- The weight of such ideas are hard to calibrate objectively, we can just go by the reasoning that newOz is "new" but still "Oz" , thus well motivated novelty shall precede over keeping the foundations intact when needed.

1.2 What is Scala ?

Scala, as it will be explained in the next section, is the main inspiration for newOz; In fact it is a double inspiration.

Scala is a **Syntactical inspiration** : which means that it syntax will be used to derive the syntax for newOz.

Scala is also a **Conceptual inspiration**,for the "new syntax thesis", alongside Elixir since like newOz they derived and were meant to be an improvement of their older versions (refs for whole section : ("A History of the Oz Multiparadigm Language" n.d.; *Scala Wiki Page* n.d.; *Elixir Wiki Page* n.d.)):

- Scala is a language based on the really popular programming Java developed by Sun Micro System, it uses the same virtual machine but Scala is truly functional (from a programming standpoint) has grow a substantial community over the years. Additionally you can compile your Scala program into a Java executable.

- Elixir is a language based on the much older, but still often updated, programming language Erlang; It has a much more modern syntax but still uses the same virtual machine. It helped a lot into growing the Erlang/BEAM (BEAM is the virtual machine) community and support.

- And thus here we have newOz, child Oz running on the same virtual machine which brings the Oz syntax much closer the one seen modern days (in fact, it isn't even a question of age but more a question of standardization since Oz was already unique for its time) like Java, C++ and others but principally Scala.

Thus the question of understanding what is Scala before using it as a reference is brought on the table. More precisely, what really matters here are the distinctive characteristics inherent to Scala which makes it a viable point of reference for creating **a new syntax for Oz** :

- Its historical common trait to newOz explained above; Not primordial for selecting it but not negligible too. Ideally, we'd like to assume that some shared roads newOz and Scala were taken when making Scala (e.g: they mostly likely wanted something modern just like us). Which, if true, would make **Scala a giant of whom the shoulders are worth standing on** instead of a random one or even nothing.

- Its proximity to many modern languages is quite important and attractive : Such proximity arguably entails that the **language is more "instinctive" for the modern user**.

- **Its proximity to Oz itself**: As functional programming languages Oz and Scala have a lot in common, and many features from Oz can find a correspondence in Scala. Such point makes the idea of using Scala or a subset of Scala (see next point) much more conceivable since, as a designer, you wouldn't need to create a syntax out of nothing when there is no bijection between Oz and Scala.

- Its relation to Ozma: see section 1.2.2 Ozma

.

1.2.1 Philosophies and Concepts differences/similarities from Oz

Since philosophies and concepts are an important field to base upon the new syntax, and since we are using Scala as the main inspiration, it is normal to compare both of them and see if something clashes or if there are new ideas to add or remove from Scala.

Let's start with the relevant similarities:

- Scala is also an interactive language, even though syntactical it doesn't have that much impact unless extreme cases.
- Scala is also a functional programming language.
- Scala also aims to be a big **modern programming language syntax**. A formal definition of a "**modern programming language syntax**" would be hard to collectively agree upon, for this thesis will use the following definition : "A modern programming language syntax is a syntax that is seen and common in the programming languages popular in the new/modern-tech industries and up-to-date schools".

Then follow by the relevant differences:

- Oz has a top to bottom hierarchy of its functional syntax while Scaladoes not.For instance in Oz put things inside others things (e.g : `{{F1 P1} P2}`) while in Scalayou chain them (e.g :`f1(p1)(p2)`) .
- Scala has many ways to write the same syntaxic component (for instance using parenthesis or brackets when you want, declaring in different ways). It is much more flexible and varied than Oz.

1.2.2 Ozma

Ozma is a conservative extension to the Scala programming language with Ozconcurrency made in the early 2000s by **Sébastien Doeraene** for a thesis concerning Oz (Doeraene, 2003).As one of the main inspiration (or more precisely, initial path) of this thesis, Ozma showed that there was a way to combine the syntax of Oz with the one of Scala (as a domain extended language). Which now separates quite distinctly Scalafrom the other candidates since discovering their compatibility with Oz would still be equivalent to losing time and taking risks. Ozma added dataflow values, light-weight threads, lazy execution and ports to Scala. Here is an small example :

```
val x: Int
val y: Int
val z: Int
thread { x = 1 }
thread { y = 2 }
thread { z = x + y }
println(z)
```

Listing 1.1: Binding variables inside threads in Ozma

Which would give this in Oz :

```
X = _
Y = _
Z = _
thread{ X = 1 }  end
thread{ Y = 2 }  end
thread{ Z = x + y } end
{Browse Z}
```

Listing 1.2: Binding variables inside threads in Oz

1.3 Motivations for a new syntax

To comply with the different goals listed previously, the main motivations for a new syntax, by priority order, are :

1. **An intuitive, user-friendly, accessible syntax** : which encompass being a modern one and simple
2. **An aesthetically pleasing syntax** : which encompass being a modern one, easy on the eyes and simple.

1.4 How to create a new syntax

Now that we know why we should make a new syntax and which path it should take, we should elaborate on the way of creating it. To do so, we have find the different ways from which we can tackle the issue and generalize them then select the most appropriate one based on their pros and cons.

Actually the process is quite natural (even though laborious); when creating a new syntax you first want to see all the tempting things available on the market. Then after seeing that there is too much information you'd to limit yourself to just the very best or the most fitting. This process of "**sources/references selection**" is what we will call the "**horizontal**" axis of the problem, where the values are languages which can be used as references or sources. We put a quotes around "**horizontal**" since it isn't actually a one dimensional axis but a multi-dimensional one (your final solution can be more or less close to one or many languages.But for the sake of simplicity, let's say that the axis is one dimensional where going left means using a variety of sources but each less and going right means using fewer sources but a bigger part of their surtaxes). Of course the designer can also add him/herself as a source and use his/her own imagination/deduction a lot or not.

After selecting your sources of inspirations, you want to truly start the conceptual side.To do so, while reusing the ecosystem analogy(but with a city),you have two choice; You can either build the city by putting gorgeous details together and trying to make them fit/harmonize iteratively/incrementally or you can first use an existing drawing the skeleton of a city, then modify it with the details to your liking.
We shall call this conception process the "vertical" axis.
In the end both issue only occurs before we have a limited amount of time and resources.You can not easily try every combinations of syntaxes from all the different languages(or even easily find them) , including the ones you could invent, and then merge them to build a perfectly fitting syntax responding to your needs.

1.4.1 Horizontal axis (few or many sources)

As introduced before, we can define our research/inspiration problem as an "horizontal" axis. Which could also be seen as a weighted sum problem equal to one (you divide your "quota" of inspiration/efforts between each sources from none to totally) : $\sum_{s_i \in sources} w_i * s_i$ where w_i is the weight we give to a given source.

During the conception of the new syntax solution based on multiple sources was taken, mainly for research sake (discovering what are the current standards) even though the "focused" sources intuition was already there.
The pros and cons using many sources ("wide research and inspiration scope") :

- Many languages have their own uniqueness or things they do better which is quite inspirational and might not always be superfluous.
- Makes it easier to not missing any trends (if you see the same pattern in multiple sources it might be a good lead).
- But makes the problem much harder
- But gives a random look to the solution unless you select only similar looking languages

The pros and cons using few sources ("narrow research and inspiration scope") :

- Makes it easier to analyse the select few languages on a deeper level.
- Make sure the final solution easily inherit the qualities of its source of inspiration.
- But you also inherit the common flaws of the sources since they won't be mitigated be a wider view- point

1.4.2 Vertical axis (bottom-up vs top-down conception)

As introduced before, once the sources are selected, we can either conceptualize our new syntax stones by stones (bottom-up) or from a drawing we took (top-down).
The pros and cons using creating a new syntax starting from each of its elements (bottom-up) :

- It is really difficult and tedious to do: For instance,you create the new syntax for your function calls, then create want to create the new syntax for something else (ex : lambdas or special features). Then your notice they don't go well together (visually or conceptually). You will then have to tweak both to make them fit, and the tweak them again when you create the new syntax for something related to one of them. In a figurative sense you advance by mainly making one step forward, then two step back. The overall problem might be exponential in term of tweaks to make if you are not highly clairvoyant (or even if you are). We could call this issue the domino effect of the bottom-up conception.
- The main and real advantage is that you end up with the syntax your truly want.

The pros and cons using creating a new syntax starting from template (top-down) :

- The template (here the syntax of the source(s) of inspiration) chosen really has to really compatible with final solution you'd expect
- Fitting the template to your needs is easier than creating one but still problematic.
- You can much more easily keep the inherent qualities of the template.

1.5 Methodology chosen for the creation of the new syntax

After elaborating options and ways to create a new syntax, we can now define the newOz has been conceived :

- It follows a mostly narrow source of inspiration of let's say 90% of Scala/Ozma (they will be grouped from now on),Oz and the rest is either novelty or really small pieces of other languages to tackle on really specific issues when needed.
- It follows a **bottom-up** conception since that solution is more stable. You know what you start with and can approximately make an assumption on what you will end up with. A top-down solution would just have been too draconian.

We expect an overall good performance from such a conceptual solutions in terms of result vs time/effort. It is still to note that, in the end, all the solution and methodologies explained in this chapter were at least explored/tested partially , some additional conclusive results can be found in the conclusion related chapter.

Chapter 2

New Syntax definition

It now important to define and motivate properly the new syntax for Oz before being able to implement and use it. Such definitions and motivations will highly be related to the ones explained in the previous chapter [1]. The main references for this whole chapter are (*Concepts, Techniques, and Models of Computer Programming* June, 2013; Seif Haridi, 2013; *Scala docs* n.d.) .

2.1 Chapter systematic structure

To reach our goal of having a well defined and motivated language, we shall take a systematic approach to this chapter and thus be sure to not leave any stone unturned.

So , for a given syntactical subject (or set of small subjects)

1. Short description of the what the syntactical subject is(for instance, "a list is ....").
2. A small code example in Scala/Ozma first since the language is more "popular" and/or easier to apprehend.
3. A small code example in Oz.
4. Concrete list of all the differences from Oz to Scala/Ozma.
5. Then the discussions and possible debate taken about this subject. The discussions are representative of the "adversarial" iterative way the new syntax was created;which consist of the writer of the thesis (me) bringing propositions and ideas to the table so that the supervisor (Mr.Van Roy) and potential external auditors (and this case, Nicolas Laurent) can discuss, find flaw and ask or propose a new solution.
6. Then a newOz example for the actual final decision.
7. Concrete enumerated and named list of all the changes (if any) from Oz and Scala/Ozma to newOz.

---

[1] It should also be noted that the real formal context-free grammar of newOz (and the languages used to created (Oz and Scala)) is in the Appendix.

8. Enumerated and named list of all motivations of all the changes cited before. Only if changes were made (to save space, refer to the choice priority otherwise).

2.2 Choices priority

From the most important to the least (but still) important criterion looked at when changing the syntax or choosing one solution over another:

1. **How well does the new syntax excerpt fits with what was already defined in newOz ?** Since everything has to be homogeneous (or in harmony), each new syntax definition should fits well with the previous ones (following the order in which they were introcuded in the thesis).(C_1**;+5pts**)
2. **Is there an almost perfect match with the language of reference Scala ?** (C_2**;+3pts**)
3. **Is there an almost perfect match with the language the current modern popular languages (as defined in the first chapter) outside of Scala ?** (C_3**;+2pts**)
4. **Can changing the syntax of this excerpt safely remove conflicts ?** (C_4**;+1pts**)

Yet, there is also a priority list disfavour (reject) once solution over another.

1. **Can this syntactical choice not be implemented under the current newOz parsing solution and resources ?** The end-goal is still to have something which can be used, if it looks better but can't easily be implemented (for instance , too many conflicts or too many changes needed) it shall be discarded. (R_1**;-100pts**)
2. **Is the solution confusing or not as natural as wanted ?** For instance declaring variables with "`val`" will be considered as confusing/not natural (these confusions are often explained in the discussion section) (R_2**;-2pts**)
3. **Does implementing this solution raise the complexity of the parsing, testing and conceptualisation significantly ?**

So , most of the time, a choice will be made by summing up the pre-made pros (C_i) and cons (R_i) to determine the final solution [2].

2.3 Value and Variables

2.3.1 Value Declaration and Use

A Value will be define as a variable which can only be assigned once.

Scala/Ozma example

```
val Y : Int
val x = 8
```

---

[2] It case of equality, the most ascetically pleasing solution wins

Oz example

```
declare
    Y
    X=8
```

Differences

- Values in Oz start with an upper case while they can be anything in Scala.
- Scala/Ozma has no declare token (uses val,def etc.. as a kind of declaration).
- Ozma has forced typing for unbound values (we should force it).
- In fact the first like in the Scala/Ozma is invalid since a value can not be unbound in the Scala/Ozma while it can in Oz

Discussions

- There is a big differences between values in Oz and values in Scala/Ozma, which is that in Scala/Ozma they want must be assigned as soon as they are declared. Which creates a difference difference between the formulation of both syntaxes since an Oz value could be unbound (equal to "_").
- The "values" in Oz are defined as "Oz variables", which might create some ambiguity if we use the val token.
- But if we used a token like var to define these "Oz variables" which can't vary (since they are only assignable once or unbound) , it would create an even more deeper difference.
- There is also the fact that the variables in Oz always start with an upper-case, which is quite different from their usual usage in other languages. We could thus use this change to allow lower case and let the parser convert them back to upper cases or escape them.
- If we use lower cases (we are talking about the first character here) for Oz variables (which also include function names or class names) , we will kind of enter in conflict with the Oz labels (which for Oz record, tuples, etc.). To avoid this there were many solutions/iterations :
    1. Oz labels could be escaped in the newOz code (which would give 'tree' for instance).
    2. The parser could dynamically detect if the user is talking about an Oz variable or Oz label and use some precedence rule if there is an ambiguity
    3. We could also get some inspiration from the Lisp programming language and use the prefix "'" before a label. Which is more natural and less complex than the previous solutions.
- Parsing lower cases into upper cases also bring a variable naming problem (for instance if you have "x" and "X" in newOz, you can't parse them "X" and "X" in newOz. There were three possible solutions :

1. The parser could try to detect that and escape than ambiguously named variable when needed.
2. The parser could escape the lower case variables (ex : newOz "`val 1 = 3`" into Oz "`` `1` = 3 ``") while keeping upper cases for those already in upper cases.
3. Or escape everything every time.

newOz final result

```
val y; // could be written without a ";"
val X = 8
```

* 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.

Final Changes list and motivations

1. **values in newOz can support both upper and lower cases**: since they can easily be parsed by keeping upper cases values as is and escaping lower case values while avoiding ambiguities with Oz labels by also changing Oz labels (next section) syntax. We can safely accept the support of lower case and upper cases which is motivated by the need of having a more widespread syntax(C_1**;+5pts**). Most people will use lower cases for their values when they have the choice (e.g : Python).

2. **The "val" token shall be used to declare a value** based on (C_1**;+5pts**), (C_2**;+3pts**) , (R_2**;-2pts**).

2.3.2 Variable Declaration and Use

The equivalent Scala/Ozma variable are Cells in Oz, they are variables which can change over the lifetime of the code execution.

Scala/Ozma example

```
var c:Int = 5 // creates the variable c with value 5
val x = c  //Returns the content of c in x
c = y //Modifies the content of c to y
```

Oz example

```
C = {NewCell 5} % Creates a cell C with content 5
X = @C %Returns the content of C in X
C := Y %Modifies the content of C to Y
```

Differences

- Variables in Scala/Ozma are declared as by the token `var` and while in Oz you have to instantiate a new Cell
- Variables in Scala/Ozma can be updated with a standard `=` operator while their values can be accessed/evaluated just by calling their names
- For Oz Cells , you have to access them via the unary operator @

Discussions

- While the variables in Scala/Ozma and Cells in Oz aren't equivalent conceptually : a variable in Scala denotes its content while in Oz it is more like C, the Cells denotes the reference of the content. But, still, it wouldn't be a stretch to associate the two.
- Changing the Oz modifier binary operator := into an `=` would be ambiguous and clash with an important philosophy of Oz of going against that. Like explained before `=` for a Cell would be either paradoxical (since it infers both side are equal) or ambiguous (for values equals really means equal while for variable it wouldn't really mean equal).
- Hiding the creation of a new cell behind the `var` token could be good and intuitive solution

newOz final result

```
var c = 5 //Creates a cell C with content 5
val x = @c  //Returns the content of c in x
c := y // Modifies the content of c to y
```

Final Changes list and motivations

1. **Cell in newOz are now instantiated like vars in Scala/Ozma** , the parser will automatically do the necessary translation (based on (C_1**;+5pts**) and (C_2**;+3pts**)). A new cell still be created the old way if needed(ex: NewCell(4)).

2.4 Data Structures Compound Types

This sections groups together the formulations of the different data types available in Oz, but before that two things are important to know.

The first one: Oz follows a fully dynamic typing system unlike Scala/Ozma which is an hybrid of dynamic and static typing. Thus, in Oz, the variable type is known when the variable is bound. The compiler will usually verify that the variable is properly used but some typing related verification can only be done at runtime.

The second one (and more important one, in the context of this thesis): Oz, as a declarative model , follows a **type hierarchy**. This when we define the syntax, features and possibilities of

given type, it should also be taken into account that its "children" will also be impacted in the exact same way.
Here is the Oz type hierarchy official definition :

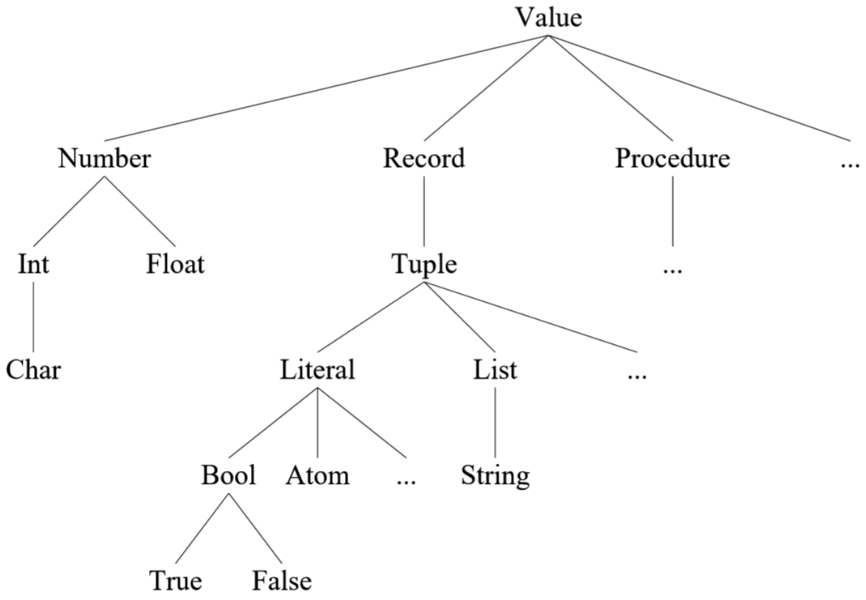


Figure 2.1: The type hierarchy of the declarative model

2.4.1 Lists

In Oz, lists more like conceptual structures than a single data type. A list is either the atom "nil",or is a tuple using the infix operator | which is given two arguments (the head of the list and its tail). Alternatively , a list could also be a string (such structure will be touched upon in the latter sections).

Scala/Ozma example

```
val L = 3::5::1::Nil
val L2 = List(3,5,1)
val L3 = L :: L2
val L4 = ::(3, ::(5, ::(1, Nil))) // more complex way to write L
```

Oz example

```
declare L L2 L3 in
L = 3|5|1|nil
L2 = [3 5 1] % equivalent to 3|5|1|nil
L3 = L | L2
```

Differences

- Scala/Ozma has the right-associative infix operator ”::” which is kind of equivalent to the ”|” **Original syntax in Oz** (used to create new immutable object or pattern matching). Here the new immutable object created is a longer
- Scala/Ozma has Nil for the tail of a list while Oz has nil.

Discussions

- A "," can be used the general parameter separator, for instance when using the original Oz syntactic sugar "[a b]" we could separate them like with comma to have a result like "[a , b]"
- The Oz list "[a b]" syntactical sugar doesn’t exists in Scala/Ozma but really practical uses. Keeping it shouldn’t be problematic since it doesn’t conflict with any philosophy or overall harmony
- Using "Nil" or "nil" for newOz is a non issue, we can just keep the one from Oz or use the one from Scala/Ozma
- The more complex way to declare lists (example L4) doesn’t seem to have any real added value.
- The object oriented way to declare lists (example L2) doesn’t seem to have any real added value.
- If we keep the token ”|” we might have issue when using the ”|” symbol else where since it is used for dijusctions (ex: x or not y) in Scala.
- Since the token ”|” is built-in **Original syntax in Oz** (you can call functions on it, like Arity), we might end up with inconsistencies if we change it.

newOz final result

```
val L = 3::5::1::nil
val L2 = [3,5,1]
val L3 = L :: L2
```

Final Changes list and motivations

1. Keeping the "[...]" syntactical sugar but adding commas to separate parameters since there is nothing controversial about it.
2. Using :: as the infix separator (based on (C_1;**+5pts**), (C_2;**+3pts**), (R_2;**-2pts**))
3. nil stays the same (based on (C_1;**+5pts**), (C_2;**+3pts**))

2.4.2 Records

In Oz, Records are structured compound entities. A closed Record has a fixed number of arguments and a label, which can be compared to "case classes" in Scala/Ozma. Case classes in Scala/Ozma are immutable data with fixed number of arguments.

Scala/Ozma example

```
case class L(a:Int, b:Int)

def main(args:Array[String]){
    var c =  L(10,10)        // Creating object of case class
    println("a = "+c.a)              // Accessing elements of case class
    println("b = "+c.b)
}
```

Oz example

```
A = 10
B = 10
C = l(a:A b:B)  % Creating a record with the label "l"
{Browse C.A} % Accessing elements of case class
{Browse C.B}
```

Differences

- In Oz records, each arguments are consist of a pair of "Feature:Field".
- In both Oz and Scala/Ozma the record/class can have their attributes accessed with the "." operator
- In Oz you aren't forced to name the features (writing $tree(IYLTRT)$ is equivalent to writting $tree(1:I2:Y3:LT4:RT)$
- In Oz you can call the function Arity to get the list of features inside the record

Discussions

- In Oz records and features start with a lower case (like atoms), thus if you wanted to call a function in newOz starting with a lower case it could be quite ambiguous.
- Records could be differentiated from function by using a different system to separate parameters (like "," for functions and a space for record). Such solution had to be given up upon since it would give some kind of randomness in the syntax but , most importantly, wouldn't solve the issue for records and functions with a lone parameter.
- After much debate, **allowing lower cases for functions and variables** seems like a important concept/philosophy in newOz. Which makes removing the ambiguity of such functionality with record/atoms a inevitable issue to tackle no matter what. The final solution is explained in the section about **Atoms** and the one about **Value and Variables**.

- While Scala/Ozma case classes and record can be grouped, using the syntax of case classes wouldn't have much sense.
- Should the "." operator be changed ? For things which were already the same in both Oz and Scala/Ozma, the answer is often **no**.

newOz final result

```
a = 10
b = 10
c = 'l('a:a,'b:a)  // Creating a record with label l
//Could also be written
browse(c.a) // Accessing elements of case class
browse(c.a)
```

Final Changes list and motivations

1. The issue with lower case features or labels was solved as explained in the **Atoms** section.
2. Keeping the "." operator ((C_2**;+3pts**),)
3. Using "," to separate parameters : mainly for consistency' sake ((C_1**;+5pts**) and (C_2**;+3pts**) with **Functions Procedures** and **Lists**).

2.4.3 Tuples

As explained the the **Records** section introduction.In Oz, it is possible to omit the features of a record reducing it to a compound term which is called **tuple**. Which makes tree (I Y LT RT) in syntactic notation for the record tree (1:I 2:Y 3:LT 4:RT) of which features are integers from 1 to the number of fields in the tuple.

While, syntactically, we can (and have to) base the new syntax of tuples from the one of **Records**, we will still have to tackle on the **infix tuple-operator** #specific to Oz. For instance 1#2 is a tuple of two elements while 1#2#3 is a tuple of three elements unlike the pair of head/tail we would get if it was the syntax for **Lists** (ex: it isn't equivalent to 1#(2#3)).

Scala/Ozma example

```
val t1 = (1,2,3) // tuple of tree elements
val t2 = (1,(2,3)) // tuple of two elements
val t3 = ()  // empty tuple
val t4 = (x) // single element tuple
print(t1._1) // printing 1st element of the tuple
```

Oz example

```
T1 = 1#2#3 % tuple of tree elements
T2 = 1#(2#3) % tuple of two elements
T3 = '#'()  % empty tuple
T4 = '#'(X) % single element tuple
{Browse T1.1} %  printing 1st element of the tuple
```

Differences

- The #operator is truly an operator in Oz
- Scala/Ozma has its own specific way (Python like) to access the fields of the tuple while in Oz treat the tuple like the record it is.

Discussions

- The tuple syntax of Scala/Ozma is way more standardized than the one of Oz
- If we were to use the syntax of Scala/Ozma it wouldn't have any conflict (syntaxically , not conceptually) with the rest of newOz
- To **stay true to the type hiearchy** of Oz , all the syntactical expressions related to **Records** have to stay the same. So using "._1" Scala / Python like notation is to be avoided.
- While both , the Oz and the Scala, tuple syntax can be compared, **they aren't the same thing** . One can be treated an operator while the other can not be. So in the absolute, **we can not replace the Oz tuple syntax with the on of Scala since they don't stand for the same logic**.
- Another solution would be to keep both of them , since the Scala syntax looks much better and it would be a shame to let it go (and letting it go might break the overall harmony). But doing so might be akin to going against the one of the philosophy of Oz which wants to minimize the redundancy.
- Maybe we could change the character for the #into something more pleasing (which couldn't be found).

newOz final result

```
// Official syntax

val t1 =  1#2#3  // tuple of tree elements
val t2 = 1#(2#3) // tuple of two elements
val t3 = '#'()  // empty tuple
val t4 = '#'(X) // single element tuple

// No Scala like Syntaxic sugar

browse(t1.1) // printing 1st element of the tuple
```

Final Changes list and motivations

1. **The original Oz syntax is kept** and still possible since there is no Scala equivalent.
2. **Not allowing the Scala syntax as a syntaxic sugar** based on ((R_1;**-100pts**), (C_1;**+5pts**))
3. Thus original Oz syntax stays the official tuple syntax.

2.5 Data type (Primitives)

This section is a follow up of the previous one related to Oz newOz data types but has its own specific structure since primitives are really small (syntactically) and used everywhere. So here the goal to debate on how those primitives should change, should look like and with what do they conflict. In fact their Scala/Ozma counter part doesn't really matter (aside from inspiration) since they are at the lowest level of Oz.

Thus, for this section only, the structure will be the following :

- List of all the possible and accepted representations of the primitive in Oz
- Exhaustive discussion on what should and can change and why
- List of all the possible and accepted representations of the primitive in newOz
- The explicit list of the decisions taken and their reasons. Whether it something change or not.

2.5.1 Atoms

An atom is a kind of symbolic constant that can be used as a single element in calculations. There are several different ways to write atoms. An atom can be written as a sequence of characters starting with a lowercase letter followed by any number of alphanumeric characters. An atom can also be written as any sequence of printable characters enclosed in single quotes.

Oz representation

Examples of Oz accepted atoms are and only are:

1. an_atom
2. theAtom
3. '### this is an atom ###'
4. 'x-oz://system/wp/QTk.oz'
5. '### this is an atom with a \'inside ###'
6. '### this is an atom with hexa asci character \x26 ###'

So, here, we are using forward quotes when we need to escape things like spaces, special characters, **the atom delimiter (straight quotes)** , and so on.

Discussions

- Atoms are used for label names (like name of records, features, etc.) so the syntax changes and needs are interlinked.
- Due to the new concept of newOz of allowing the 1st case to be a lower case for variables we have to fiend a way to differentiate Oz variables and atoms.
- One of the solution could be getting some inspiration form the Lisp language (*Lisp variable official doc* June, 2020) and prefixing atoms with a '(straight single quote) character (ex: **'something** and **´something´** would be atoms while **something** would be a variable. It is important to notice that the character is an **straight single quote** and not a forward quote since it would allow us to keep the forward quotes for escaped atoms.
- Keeping all the logic related to characters inside a delimited atom in newOz was a question (could make the parsing more complex).

newOz final representation

Examples of newOz accepted atoms are and only are:

1. 'an_atom
2. 'theAtom
3. ´theAtom´
4. ´### this is an atom ###´
5. ´x-oz://system/wp/QTk.oz´
6. ´### this is an atom with a \´ inside ###´
7. ´### this is an atom with hexa asci character \x26 ###´

Final Changes list and motivations

1. **The syntax of non delimited atoms starting by a lower case was removed**. The main idea here is to leave that syntax solely to Oz variables (values names, functions names , classes names).
2. **A new unary operator ' (single straight quote) was added** to declare atom easily in newOz. It should be directly attached to the wanted atom and and that atom can only follow this regular expression : [a-zA-Z](\w|\d)*
3. **The forward quotes escape delimiters is kept** since there no real reasons to not to.
4. **Those delimiters can still include escaped characters or special notations like hexas** since there is no real reasons to not to (parsing complexity aside).

2.5.2 Boolean and their associated operations

Booleans (logical values or expression which are either **true** or **false**) are a subject easy to tackle since they are widely used and the **true** or **false** symbols are the same in Oz and Scala/Ozma (in both case have a high lvl of precedence, so **true** will always be a boolean and not an atom, function, variable or what not).

The real issue here will be to redefine their related logical operators and maybe even see if missing ones could be added.

Oz representation

1. **true**
2. **false**
3. **<exp> andthen <exp>** : equivalent to **<exp>** && **<exp>** in Scala. It evaluates the second expression only if the first one is true, returns a Boolean.
4. **<exp> orelse <exp>** : equivalent to **<exp>** || **<exp>** in Scala. It evaluates the second expression only if the first one is false, returns a Boolean.

Don't forget that the **<exp>** could be a pattern or expression which can be evaluated as a Boolean.

Discussions

1. The & symbol used in Scala/Ozma is already used in Oz for Oz character: for instance "&*character* is equivalent to "97".
2. The | symbol is already used in Oz for lists.
3. Scala has also the logical (bitwise) symbols & and | used to evaluate both sides. They are not available in Oz unless you create them.
4. The new parser could come with a library allowing the bitwise operations.
5. **true** and **false** starting with lower cases is not problematic thanks to their precedence.

newOz final representation

1. **true**
2. **false**
3. **<exp>** && **<exp>**
4. **<exp>** || **<exp>**

Final Changes list and motivations

1. **andthen is changed by "&&"** while the Oz unary operator symbol "&" is changed with the unused character degree "°" to avoid conflicts.
2. **orelse is changed by "||"** while the Oz list infix symbol "|" is changed with the Scala list operator ":: to avoid conflicts (more details in **Lists** section).
3. No specific actions were taken for the bitwise "|" and "|" since they aren't that important (functions for them can be quickly written when needed).

2.5.3 Strings and Virtual Strings

A string is a list of character codes, written and used mostly the same way in both Scala and Oz

Oz representation

1. "this is a string"
2. [79 90 32 51 46 48]
3. [&O &Z & &3 &. &0]

The three ways are equivalent (notice that an Oz character can be an empty space).

Discussions

No real discussion aside from aligning with the other sections.

newOz final representation

1. "this is a string"
2. [79, 90,32,51,46,48]
3. [°O,°Z,° ,°3 ,°. ,°0]

2.5.4 Numbers

Numbers are either integers or floating point numbers. Their syntax and representation won't change much from the original Oz in newOz.

Oz representation

1. 314, 0,~1 for integers
2. 1.0, 3.4, 2.0e2, and~2.0E~2 for floating points number

Discussions

1. The only real discussion available here is whether or not to keep the "~" symbol for negative numbers.

newOz final representation

1. 314, 0,~1 for integers
2. 1.0, 3.4, 2.0e2, and~2.0E~2 for floating points number

Final Changes list and motivations

1. Things were kept as they were for the sake for simplicity ((C_1;**+5pts**) , (C_2;**+3pts**), (C_4;**+1pts**))

2.6 Functional Syntax

2.6.1 Functions Procedures

In Oz, procedures and functions are more or less the same things. For instance you can write an anonymous procedure like :

1. <x> ¯ proc{**$** y_i ... y_n } <s> **end** : where x is the procedure value (since procedures are a type value see the **the type hiearchy**) and **$** indicates the anonymity of the procedure (see Lambdas section, the next one) .
2. proc{<x> y_i ... y_n } <s> **end** : is a named procedure with the name being <x>.

Functions have the same overall syntax but return values instead.

Scala/Ozma example

```
def fact(N) = {
  if (N==0) 1
  else N *fact(N-1)
}

print(fact(10))
```

Oz example

```
fun {Fact N}
  if N==0 then 1 else N*{Fact N-1} end
end

{Browse {Fact 10}}
```

Differences

- Variable in Oz are in upper case while they can be anything in Scala (see Value and Variables section for more details) .
- Function are called in a completly different way in Scala/Ozma than Oz. Instead of being between curly brackets with the function name on the left side and its parameters on the right side; Scala/Ozma uses a more classical notation with name of the follow followed by the parameters between parenthesis.
- In fact the first like in the Scala/Ozma is invalid since a value can not be unbound in the Scala/Ozma while it can in Oz
- The inside of the function is delimited by curly brackets in Scala/Ozma while in Oz with just have **end** to signify the end of the function.
- In Scala there is an ¯ between the function's signature and its code block.

Discussions

1. Would the ¯ in the signature have any real value in newOz ?
2. How can we support lower name for functions ? (answer in the Value and Variables section).
3. Should the way we call function change, what does it imply ?
4. If we call functions like in Scala (ex : `print(x)`), chaining functions returning functions would give a completly different result and parsing . For instance, `{{F1 P1} P2}` in Oz would become `f1(p1)(p2)` in newOz . We'd go, when writing function calls, from an imprecated visualisation to something more classic akin to chaining (which technically would be more exact).
5. Should we separate parameters with a comma like in Scala/Ozma ? There doesn't seem to be any downside to it.
6. Scala doesn't syntactical differentiate functions and procedures. If the use the keyword **def** for both we might have a slight problem of ambiguity but more importantly loose their explicitly different annotation (**fun** and **proc**) which brought some education value.

newOz final result

```
def fact(n) {
  if (n == 0){ 1 }
  else{n * fact(n-1)}
}

browse(fact(10))
```

Final Changes list and motivations

1. **The keyword fun becomes** . to align with Scala
2. **The keyword proc becomes defproc** to keep be still able to convey their differences syntactically but also avoid useless potential parsing issues.
3. **Parameters are separated with a comma** ((C_1;**+5pts**) , (C_2;**+3pts**)).
4. **Functions calls are now done like in Scala** ((C_1;**+5pts**) , (C_2;**+3pts**), (C_4;**+1pts**))

2.6.2 Lambdas

As introduced before, lambdas are "just" anonymous functions or procedures.

Scala/Ozma example

```
// lambda expression
val ex1 = (x:Int) => x + 2

// with multiple parameters , brackets can be used
val ex2 = (x:Int, y:Int) => { x * y }

println(ex1(7))
ex2(2, 3)
```

Oz example

```
%lambda expression
Ex1 = fun {$ X } X + 2 end

%with multiple parameters and as procedure
Ex2 = proc {$ X Y} X * Y end


{Browse {Ex1 7}}
{Ex2 2 3}
```

Differences

- In Scala , lambdas can have code blocks or not.
- Scala doesn't differentiates functions and procedures.
- Parameters are separated by a comma in Scala.

Discussions

1. Do we need to force code block or do both like in scala ?
2. Would the right arrow just be decorative if used for newOz ?
3. The right arrow binary operator arguably conveys well the idea of inputs on the left side and outputs on the right side.
4. Do we need two kind of lambdas to differentiate functions and procedures ?
5. Why not use the meaning of the right arrow to differentiate lambda procedures and lambda functions (since function returns something, a right arrow makes sense).

newOz final result

```
//lambda function expression
ex1 = (x) => {x + 2}

//with multiple function parameters and as procedure
ex2 = (x, y) => {x * y}

//lambda procedure with multiple parameters and as procedure
ex2 = (x, y){browse(x,y)}

browse(ex1(7))
ex2(2, 3)
```

Final Changes list and motivations

1. **Lambdas will use the $\Rightarrow$ binary operator** to separate the anonymous function signature and code block. It was mainly decided based on aesthetics and the natural "feeling" when writting item. So mostly arbitraly.
2. **The $ symbol is not explicitly written anymore**. Mainly because the syntax of function/procedures and lambdas now deviates (due to the arrow in part).
3. **Lambdas with an arrow will always be treated as functions** ((C_1;**+5pts**) (C_2;**+3pts**),(C_4;**+1pts**)).
4. **Lambdas with no arrow will always be treated as procedures**((C_1;**+5pts**),(C_4;**+1pts**)).
5. **newOz doesn't allow to write a lambda without code block**((R_1;**-100pts**)(R_2;**-2pts**), (C_2;**+3pts**)).
6. **The parameters are separated by a comma**((C_1;**+5pts**) (C_2;**+3pts**)).

2.7 Classes and Objects

2.7.1 Inheritance

Before elaborating on classes and objects, it might be better to first get familiar with the syntax and concepts of inheritance (more specifically in the context of Oz).

Classes may inherit from one or several classes in Oz. The main differences with languages like Scala or Java is that, in Oz,they can also call any direct or indirect parent by specifying its name. Alongside the lack of keywords such as **super** or **this**.

Scala/Ozma example

```
class ListC extends BaseObject { ... }
```

Oz example

```
class ListC from BaseObject .. end
```

Differences

- Scala has **extends** while Oz has **from**
- The different parents are separated by a comma in Scala instead of a blank space in Oz
- Class names can't be written with a lower case in Oz
- There is a code block in Scala

Discussions

- Not too many things to discuss since the syntax is pretty similar once you apply all the newOz principles already seen (code blocks, variable names , parammeters).
- The token **from** could be changed by **extends**

newOz final result

```
class ListC extends BaseObject,BaseOject2 { ... }
```

Final Changes list and motivations

1. **The token from is changed into extends** to align with Scala but also because it is more widely spread for that scenario.
2. **Parent classes are separated by a comma** to stay aligned with the rest of the newOz syntax.

2.7.2 Classes and Objects

Scala/Ozma example

```
class Counter{
    var value:Int
    val pm:Unit = privateMethod
    def browse() ={
      print(value)
    }
    def init(Value) ={
      value = Value
    }
    def inc(Value) ={
      value = value + Value
    }

    private def privateMethod(val X:Int): Unit =
    {
        print(X)
    }
    def publicMethod(): Unit =
    {
        this.privateMethod(5)
    }
}

class Child extends Counter{
   def  childMethod() : Unit = {
        super.pm(5)
   }


  def superCall() = { super.inc(1) }
}
```

Oz example

```
class Counter
    attr val
         pm:PrivateMethod
         % making the private method as an
         % attribute so it can be accessed by the children
    meth browse
      {Browse @val}
    end
    meth inc(Value)
      val := @val + Value
    end
    meth init(Value)
      val := Value
    end

    % Upper case mean private
```

```
    def PrivateMethod(X)
      {Browse X}
    end

    % Method with no parametters
    def publicMethod
        {self PrivateMethod(5)}
    end
end

class Child from Counter
  meth childMeth L=@pm in {self L(5)} ... end

  meth superCall Counter,inc(1) end
end
```

Differences

- Private methods are defined by a starting upper case on their name in Oz while they have `private` prefix in Scala.
- In Oz the "super" (parent's method call) allows you to call the method of any parent class your class inherit from directly (direct parent) or indirectly (parent of parent ...) by using its class name followed by a "," then the method. In Scala this is just a `super.parentMethod (...)` call for the direct parent
- In Oz , attributes have to be declared first.

Discussions

- Discussions related to the methods are in the next section
- A keyword **feat** was available in Oz but won't be supported in newOz since it doesn't have much value. Feats were stateless components meant to be used like in a record.
- The **attr** followed by a list of attributes (cells) is quite different from the Scala syntax where the val keyword is before each individual attributes.
- Being able to call directly any specific parent is a cool feature of Oz but quite costly in term of syntax appeal.
- The way to call methods of specific parents (ex: ParentName,parentMethod) was really counter intuitive, maybe it could use something like **super** instead.
- Other things like method names , variables names , code blocks were already discussed in the previous sections.
- **self** keyword can easily changed by **this** ((C_2;**+3pts**) , (C_4;**+1pts**)).
- Creating a **super**() keyword function in newOz just via the parser might be hard since we would need to detect the specific parent via code analysis. A solution between the two might be found (like **super**(parentName).methodeName()).

- If there is only 1 direct parent, the parser could easily find its name. So we can allow to write **super**.methodNam() and **super**().methodNam() only if there is one parent.

newOz final result

```
class Counter{
    attr v,x;
    prop z
    attr pm = privateMethod
    // Order of proprieties and attributes don't matter
    def 'myBrowse() {
      browse(@v)
    }
    def 'inc(Value) ={
      v := @v + Value
    }
    def 'init(Value) ={
      @v := Value
    }

    def PrivateMethod(X) {
      Browse(X)
    }

    def publicMethod() {
        this.privateMethod(5)
    }
}

class Child extends Counter{
   def  childMethod(){
        var L = @pm
        this.L(5)
   }

   def superCall(){
        super(Counter).inc(1)
   }
}
```

Final Changes list and motivations

1. **self keyword is changed into this** to align with Scala but also because there is no conflict.

2. **Parent calls are now done via the super operator** always taking an expression in parameter(ex : **super**(MyOnlyParentName).methodNam()). ((C_1**;+5pts**), (C_2**;+3pts**) , (C_4**;+1pts**))

3. Attributes, proprieties and methods can now be declared in any order ((C_1**;+5pts**), (C_2**;+3pts**) , (C_3**;+2pts**)).

2.7.3 Methods definition

The section is specific to Oz method head and signatures, and is thus treated differently. The original version and the newOz are listed before global discussion.

Method definitions list

- **Fixed argument list** :
  - **Original syntax in Oz**:

```
meth foo(a:A b:B c:C)
    % Method body
end
```

  - **Adapted syntax in newOz** :

```
def 'foo('a:a, b:B, c:C){
    // Method body
}
```

- **Flexible argument list**:
  - **Original syntax in Oz**:

```
meth foo(a:A b:B c:C ...)
    % Method body
end
```

  - **Adapted syntax in newOz** :

```
def 'foo('a:a, b:b, c:c, ...){
    // Method body
}
```

- **Variable reference to method head**:
  - **Original syntax in Oz**:

```
meth foo(a:A b:B c:C ...)=M
    % Method body
end
```

  - **Adapted syntax in newOz** :

```
def 'foo('a:a, b:b, c:c ...)=M{
    // Method body
}
```

- **Optional argument**:
  - **Original syntax in Oz**:

```
meth foo(a:A b:B<=V)
    % Method body
end
```

  - **Adapted syntax in newOz** :

```
def 'foo('a:a, b:b<=V){
    // Method body
}
```

- **Private method label**:
  - **Original syntax in Oz**:

```
meth A(bar:X)
    % Method body
end
```

  - **Adapted syntax in newOz** :

```
def A('bar:X){
    // Method body
}
```

- **Dynamic method label**:
  - **Original syntax in Oz**:

```
meth !A(bar:X)
    % Method body
end
```

  - **Adapted syntax in newOz** :

```
def !A('bar:X){
    // Method body
}
```

- **The "otherwise" method**:
  - **Original syntax in Oz**:

```
meth otherwise(M)
    % Method body
end
```

  - **Adapted syntax in newOz** :

```
def 'otherwise(m){
    // Method body
}
```

Final Changes list and motivations

Here most changes were due to a direct impact of all the decisions taken until now. Only some specific points are new and need to be motivated :

1. **The keyword meth was change to the keyword def** since there is no read problem with doing that and it also brings the syntax closer to the one of Scala
2. **Private methods are still used with an upper case**
3. **Public methods are now use a atomLisp** (e.x "def 'inith()").
4. These change there since other wise the parser couldn't differentiate private and public methods when called outside of the class definitions since the newOz is currently stateless (which would have given that solution a (R_1**;-100pts**))

2.8 Conditionals

2.8.1 If,else, ternary

Scala/Ozma example

```
val L = List (3 , 5, 1)
if (L.contains(3)) println("has 3")
if (L.contains(3)){
  println("has 3")
}else if(L.contains(5)){
  println("has 5 but not 3")
}else{
  println("has no 5 and no 3")
}
```

Oz example

```
declare  L in
L = 3|5|1|nil
if {Contains{ L 3}} then {Browse "has 3"} end
if {Contains{ L 3}} then
   {Browse "has 3"}
elseif{Contains{ L 5}}
   {Browse "has 5 but not 3"}
else
   {Browse "has no 5 and no 3"}
end
```

Differences

1. Scala's **if** is always followed by condition inside parenthesis
2. Oz conditional blocks only need 1 keyword end at the end

3. An if doesn't need a block if it covers only 1 statement
4. Both syntaxes already include easy ternary operator like syntaxes
5. Since there is nothing specific in Ozma/Scala conflicting with Oz, the whole grammar can be kept

Discussions

1. The forced parenthesis aren't a must (like in Python for ,instance).
2. Forced code blocks aren't bad, in fact Oz was already kind of like that.
3. **elseif** and **else if** debate doesn't matter much, with **else if** you might feel like you are doing a **if** after inside an **else**.

newOz final result

```
var l = [3 , 5, 1]
// ifs have to be followed by a code block
if (Contains(l,3)){browse("has 3")}
if (Contains(l,3)){
  browse("has 3")
}else if(Contains(l,5)){
  browse("has 5 but not 3")
}else{
  browse("has no 5 and no 3")
}
```

Final Changes list and motivations

1. Following Scala syntax but forcing code blocks even for single line statement since it is much easier to parse with the current parser.

2.8.2 Switch cases

A parallel conditional or "switch case" is a syntactic well to abbreviate a tedious list of **if** and **elseif**.

Scala/Ozma example

```
def test(X:Any):Unit = {
    X match {
      case 1  => println("Case 1")
      case 2  => println("Case 2")
      case 3 && true => println("Case 3")
      case 4  => println("Case 4")
      case 5  => println("Case 5")
      // catch the default with a variable so you can print it
      case whoa  => println("Unexpected case 6 : " + whoa.toString)
```

```
      // or case _ => println("Case default")
    }
    (X : @switch) match {
      case 6  => println("Case 6")
      case _ => println("Case default")
    }
}
```

Oz example

```
proc {Test X}
    case X
    of a|Z then {Browse   case  (1)}
    [] f(a) then {Browse   case  (2)}
    [] Y|Z andthen Y==Z then {Browse   case  (3)}
    [] Y|Z then {Browse   case  (4)}
    [] f(Y) then {Browse   case  (5)}
    else {Browse   case  (6)} end
end
```

Differences

1. **@switch** keyword for Scala tableswitch compile time optimization has no use or equivalent in Oz

2. The overall form of both syntaxes are quite different but the logic stays the same: we try to match a list of patterns to a value and execute the code block when there is a match

3. Scala and Oz switches are thus pretty much compatible from a functional standpoint aside from the default case with variable.

4. Oz doesn't really need the default case the default case with variable (you will always know it)

Discussions

1. Not much discussions since the syntax have a lot of equivalences.

2. We don't need to force code blocks for the cases since there won't be any problem during the parsing

3. The **match** keyword position in Scala isn't that practical, putting the subject after the **match** is easier to parse. The only down size is that it doesn't read as well (ex: "x match something" is more natural than "match x something").

newOz final result

```
defproc test(X) {
    match X {
      case 1  => browse("Case 1")
      case 2  => browse("Case 2")
      case 3   => browse("Case 3")
      case 4  => {browse("Case 4")} // Switchs can have blocks
        // A block a equivalent to writting "local in" if there  // is a
    val/var declaration inside
      case Y::Z && Y==Z => browse("Case 5") // pattern matching on a list
      // catch the default with a variable so you can print it
      else browse("Unexpected case 6 : ", "whoa")  // else can have blocks
    or not
     }
}
```

Final Changes list and motivations

1. **The "[] <Pattern> then <Expression> end" notation used for cases is changed into "case <Pattern> then <Expression> ⇒ [] notation"** : since the one Scala is much easier to read for humans.
2. **The "caseX of...end" Oz notation will be changed into "match X { ... }"** where the first **case** pattern will of the **of** pattern in Oz
3. **The "cond .. of" syntax will be the same but with a cond instead of case**. Since both syntax where already equivalent in Oz.

2.8.3 Exceptions

Exceptions can also be considered as conditional scenario: you have a code you want to execute in its try block, and many issues can be raised to be caught in the catch block to finally, maybe, run a specific code once all is done (like closing a file).

Scala/Ozma example

```
try {
    // your scala code here
}
catch {
    case foo: FooException => handleFooException(foo)
    case bar: BarException => handleBarException(bar)
} finally {
    // your scala code here, such as closing a database connection
    // or file handle
}
```

Oz example

```
try
    % your Oz code here
catch  P1 then S1
    [] P2 then S2
    [] P3 then S3
finally
    % your Oz code here
end
```

Discussions

1. Since, in Oz, the catch expressions/statements follow a "pattern then expression/statement" schema, just like switch case. A syntax close to the switches could be elaborated.
2. Coincidentally, or not, Scala also has the same similarity between its switches and catches.
3. The try + code block Scala way of writing exception is nice and quite compatible with Oz's.
4. Same for the finally statement or the raise expression.
5. Naturally in the newOz we can't have an else since it is used for finally

newOz final result

```
try {
    // your newoz code here
    E1
}
catch { // optional catch
    case P1 => S1 // at least one default case
    case P2 => S2
    case P3 => {S3} // with or without block , need a block for //
   declaring as  a local
} finally { // optional finally
    // your newoz code here, such as closing a database connection
    // or file handle
    S4
}
```

Final Changes list and motivations

1. **try + code block** will be the way to write the try schema.
2. **It can be followed by a catch with a series of case + pattern => block-code inside,like in Scala** since the original Oz can easily be converted in the Scala like syntax when we case ourselves on the newOz **switch** syntax.
3. **The followed by a finally + code block**

4. **And raises expressions are now the raise + code block** since it is the self-evident solution in term of alignment.
5. **No else in the catch block** naturally.

2.9 Locks and threads

2.9.1 Threads Locks

Oz allows the programmer to easily create new many threads and execute the code inside them inside other processes. In fact, the thread syntax of Oz was already much different from the other syntax seen in the language since closer to a modern one. So its comes of as a nice surprise that it almost doesn't need to be changed at all (code blocks have to be adapter). While there will be no change the subject still has to be "threaded" upon.

Scala/Ozma example

```
println("Main thread")
thread { println("New light-weight thread") }
thread(println("New light-weight thread 2"))
println("Continuing main thread")
```

Oz example

```
{Browse "Main thread"}
thread {Browse "New light-weight thread"} end
thread {Browse "New light-weight thread 2"} end
{Browse "Continuing main thread"}

%% a lock
lock L then J in J=@C C:=J+1 end
```

Differences

- In Scala/Ozma , curly brackets and parenthesis are often interchangeable.
- There is no real equivalent between Scala/Ozma/Java locks and Oz.

Discussions

- We don't need to keep the Scala idea that curly brackets and parenthesis can be interchanged most of the time. It would create a redundancy we don't want.

newOz final result

```
browse("Main thread")
thread{browse{"New light-weight thread 2")}
browse("Continuing main thread")
// a lock with an expression as parameter
lock(L){val J=@C; C:=J+1 }
// lock with no param
lock{val J=@C; C:=J+1 }
```

Final Changes list and motivations

- Threads syntax aligned on Ozma ((C_1;**+5pts**), (C_2;**+3pts**),(C_4;**+1pts**)).
- Lock syntax aligned with everything already defined ((C_1;**+5pts**), (C_2;**+3pts**),(C_4;**+1pts**)).

2.10 Built-In functions and functors

2.10.1 Functors

Scala/Ozma example

```
import org.scalatest.flatspec.AnyFlatSpec
```

Oz example

```
functor
import
    Browser
    FO at file :///home/mydir/FileOps.ozf
export
    Browser

define
    {Browser.browse {FO.countLines /etc/passwd }}
end
```

Differences

As it can be seen, Scala/Ozma don't really provide a solution comparable to Oz. In fact, this is one of the few case where if we want some template with have to look at the syntax of languages with keywords import or export like Python for instance. So technically, almost everything is different.

Discussions

- The imports statements could be repeated like in Scala (with one import line of each variable to import)
- If we did it we would have to do the same for exports.
- While it would look better for imports, it would also counterproductive become cumbersome for exports (since the values expected for exports are quite shorts , doing a new line for each would be tedious).
- The export could be written as export followed by its values separated by commas.
- The functor keyword is not might not look necessarily needed but without it an oz conversion might be difficult since it is possible to write a functor without import nor export in Oz

newOz final result

```
functor
import Browser('browse:browse)
export X
import FO from   file   :///home/mydir/FileOps.ozf
{
   browse(FO.countLines(   /etc/passwd  ))
}
```

Final Changes list and motivations

- **A new import line has to be written for each import** : ((C_1;**+5pts**), (C_2;**+3pts**), (C_3;**+2pts**))
- **A new export line has to be written for each export** ((C_1;**+5pts**), (C_2;**+3pts**), (C_3;**+2pts**))
- **The functor code block can be empty**
- **The functor code block can't be missing**
- **Order doesn't matter anymore for imports and exports**((C_1;**+5pts**), (C_2;**+3pts**), (C_3;**+2pts**), (C_4;**+1pts**))

2.10.2 Built-In function list

As seen above the often used "browse" function was easily redefined with a lower case with no help from the parser. It can work well for functors since the user is the one naming them , but for the one built in they will either have to be hard-coded in thenewOz parser or writing like with an uppercase. The most used ones like "new(...)" and "newCell(...)" are supported by the newOz parser.

2.11 Others

2.11.1 Comments

Discussions

Since Oz supports both single line and multi line comments, we just need to adapt them to fit those from Scala.

Final Changes list and motivations

1. **Single line comment can be written with two forward slashes** , to align with Scala but also free the Oz comment symbol and use it for modulo
2. **Multi line comments can be written with with same delimiters used in Scala** : the feature is a nice addition and not that code breaking.

2.11.2 Local in

"Local ... in ... end" is equivalent to a code block in newOz, the newOz parser will do the conversion of code blocks into a local statement/expression if there are any values or variables declared inside (at the beginning of the code block only , just like in Oz). This is a 1:1 translation thus an easy choice to make ((C_1;**+5pts**) , (C_2;**+3pts**) , (C_4;**+1pts**)) .

2.11.3 Declare in

In newOz the interactive keyword "declare" can be be put before a code block to act like a "declare" in Oz. This choice overall fits quite easily with everything which was already defined ((C_1;**+5pts**) , (C_2;**+3pts**) , (C_4;**+1pts**)) .

2.11.4 Note

If anything wasn't cited in this chapter (like a small operand) , it is probably defined in the appendix often under the reasoning (C_1;**+5pts**), (C_2;**+3pts**) and (C_4;**+1pts**) or maybe just not officially supported in the (*Concepts, Techniques, and Models of Computer Programming* June, 2013) book of reference for this chapter.

Chapter 3

New Parser/Compiler and its implementation

After the conception and elaboration of the newOz syntax, we now have to make it usable and available to the user. Concretely, a newOz programmer should be to write a program accepted by the newOz syntax, compile it and run it **conveniently** on most computers. **That newOz compiled program should also work the same as its Oz equivalent** (for instance, not having different results or execution threads).

3.1 Parser and Compilers theory recap

Let's quickly recapitulate what both, a parser and a compiler are since they are what we will need to make newOz usable.

3.1.1 What is a parser ?

A parser is a software component that takes input data (a text in our case) and builds a data structure (an abstract syntax tree in our case). That hierarchical structures allows the parser to compare the input (or a part of it) to its root node, then its children if any, recursively until the whole input is matched (*Parsers Wiki Page* n.d.).

3.1.2 What is a compiler ?

[1] A compiler is a computer program that translates computer code written in one programming language into another language. The name compiler is primarily used for programs that translate source code from a high-level programming language (newOz or Oz) to a lower level language (for instance Assembler) to create an executable program (*Compiler Wiki Page* n.d.).

So the goal here is to accept/parse newOz code and compile it into an executable program.

---

[1] The newOz parser might be referred as a compiler depending on the view points (when decorating the Oz compiler for instance)

3.2 Available solutions

3.2.1 Rewriting/Modding the Oz compiler

We could rewrite,modify or extend the original Oz compiler and parser, which would truly make newOz the "new" Oz. But the difficulties of such solution are many:

- Takes much more time than the next solution
- Many issues could pop up and aren't predictable due to using an already existing code.
- Extensive testing and regression testing would be needed.

3.2.2 Creating an newOz → Oz parser

The other solution (which is the chosen one) consist of creating a new parser in a modern environment (here Scala 2.12) , parsing and translating the newOz into an Oz before compiling it with the original Oz compiler.

The advantages are much appreciated:

1. Easier to do than rewriting the Oz compiler.
2. Easier to read since no legacy but also using modern solutions.
3. Can easily create different variations and iterations of the same parser (by using class inheritance or case classes).

For a few disadvantage:

1. Harder to print and notify the user of the different exceptions and errors (since your translator would only be a middleman).
2. Has to be hidden in an efficient way from the user, so that it feels like the normal Oz compiler.

3.2.3 Parser used Implementation

Seeing that we are going with a parser + translator solution under a modern language. We have to base ourselves on a specific parser library to implement/extend instead of starting reinventing the wheel.

Thus, the parser library chosen here is an open-source one named "**Scala Parser Combinators**"(*Scala Parser and Combinators doc* n.d.) which was library originally built-in and used by Scala itself. This generic parser combinators works by matching regular expressions (called **Parser**) with specific relationships between each others. So literally combining passers together. For instance, a simple terminal (as in not encompassing another parser inside) parser could be written as :

```
def number: Parser[Int]    = """(0|[1-9]\d*)""".r ^^ { _.toInt }
```

Which basically accepts integers.

Then we can define a simple binary expression by creating another parser which would combine numbers and symbols together for instance :

```
def twoTermDivisionExpression: Parser[String] = number ~ "+" ~ number
     ^^ { case n1 ~ addSymbol ~ n2 => "" + number + " sumed up with " +
   numer  }
```

Which would accept two numbers separated by a "+" (the ~ means "followed by") and translate the "+" into "summed up with" to return worded version of such addition.

Naturally, **Scala Parser Combinators** has many ways to combine parsers and allows much more complex parsing and mapping. The four main parser combinators are :

- p1 ~ p2 (**sequencing**) : must match p1 followed by p2. The "~" itself hides to a separating regular expression named "whitespace".
- p1 | p2 (**alternation**) : must match either p1 or p2, with preference/precedence given to p1.
- p1.? (**optionally**) : may match p1 or not.
- p1.* (**repetition**) : matches any number of repetitions of p1

These four operators are enough to parse and accept most of our grammar.

3.3 Issues encountered

3.3.1 Keeping code format

While the intermediate generated Oz source isn't really meant to be read by humans, it is still important to have to keep it clean and well written for multiple reasons (ex: debugging, showing translation examples, avoiding errors). It is thus necessary to translate the newOz into an human readable intermediate Oz before compilation.

To do so, we have to have explicitly create a system for it since the **Scala Parser Combinators** (and most passers) retain the original formatting due to ignoring it during the parsing. Two solutions are thus envisaged :

1. **Pretty printing**
2. **Parsing everything**

Pretty printing

Pretty printing refers to the way most integrated development environments (IDE) allow their user to reformat the code according to a predefined format and set of rules. Like for instance :

- The maximum length of code lines.
- When to break into a new line (for instance, before or after a block).
- How to separate different syntactical elements (like a list of parameters).

These kind of rules could be "hard-coded" in the new parser and thus produce a reliably "pretty" Oz source code. The main advantage here is the relative ease of implementation, since we know where we are in the code during the parsing we can just format the code on the go.

But the major downside is that most of the time the translated code will be quite different from the original code in term of format. Which is problematic when trying to read it and compare it to the original one. For instance if the Oz compiler says there is an error at run-time on line 45,12th character, inside the translated code; You'd want to find the same error at the same place in the original code (more or less, at least) so that the user won't have to read the translated files instead of the original.
This is why the solution in the next section was the one which was selected.

Parsing everything

Like said previously, usually, **Scala Parser Combinators** (and most parses) skip a set of predefined white spaces and tabulation which allow the user to have much less rigidity when writ ting his code. He/She can thus format the code in a good, human readable way.

It also helps a lot when making the parser by making its implementation much less complex since you won't have to always focus making sure your parser accepts a formatting as flexible as possible.

But to solve our issue of keeping the translated code as close as possible to the original code, we can get rid off this convenient feature (via the **skipWhiteSpace**= false option of **Scala Parser Combinators**) and do an all encompassing parser.

Let's look at an example before discussing the advantages and limitations of such solution. To do so we shall take a moderately difficult example of parsing and translating an **if else**.

Example

1. Let's first give a **if else**newOz code to our parser :

```
if(p1){
    s1
}else if(p2){
    s2
}
```

2. Which would be matched should be matched with as **statement** → **ifStatement** in our parser.

3. The code translating it is the following;

```
def ifStatement: Parser[Any] = `ifOpt` ~ betweenParExpression ~
 openningBlockOpt ~ inStatement ~ closingBlockOpt ~ elseIfStatement
 .* ~ elseInStatement.? ^^ {
  case ifOp ~ exp ~ oBO ~ inExp ~ cBO ~ eIfList ~ optElse =>
    translate(ifOp, exp, oBO, inExp, eIfList, optElse, if (optElse.
 isEmpty) null else null, cBO)
}
```

   Remember that is a **Scala Parser Combinators** wrappers equivalent to saying "x y => matched character sequence x should be followed by character sequence why" .

4. So, in the code above, as you can see , the **Scala Parser Combinators** which stands for the original separation/formatting can't be retrieved when trying to translate our **if else** statement.

5. Now, if we implement our solution by adding an explicit regular expression for separator we'll name "sep" ("**(\s|\n)***") and disable the **skipWhiteSpace** feature (which will make the not wrap a separation anymore) of **Scala Parser Combinators**. We would now be able to retrieve and use the original formatting.

6. And get the following code :

```
def ifStatement: Parser[Any] = `ifOpt` ~ sep ~ betweenParExpression
  ~ sep ~ openningBlockOpt ~ sep ~ inStatement ~ sep ~
 closingBlockOpt ~ sep ~ elseIfStatement.* ~ sep ~ elseInStatement.?
  ^^ {
  case ifOp ~ sep1 ~ exp ~ sep2 ~ oBO ~ sep3 ~ inExp ~ sep4 ~ cBO ~
 sep5 ~ eIfList ~ sep6 ~ optElse =>
    translate(ifOp, sep1, exp, sep2, oBO, sep3, inExp, sep4, eIfList
 , sep6, optElse, if (optElse.isEmpty) null else null, cBO)
}
```

   Where even separations are taken into account and potentially translated.

7. To get the mostly equivalently formatted Oz translation :

```
if`p1`then
     `s1`
elseif`p2`then
     `s2`
end
```

Evaluation of the solution

The final solution will be a mix of both:mainly full parsing aided by pretty printing to fix some of its limitations.

Even though we end up with a relatively well formatted code, this technique still has many limitations:

- The keywords in Oz and newOz aren't always the same length.

- Some formatting loss is inevitable due to the fact that newOz often hashas additional characters (mainly brackets).
- The solution comes with an implementation overhead (adding "sep" everywhere but also harcoding some pretty print)

Those limitations are, in the end, still affordable and don't go against the usefulness of our solution.

3.3.2 Managing precedence

The parser (and newOz) have to manage two kind of precedence:

1. **The operators/symbols precedence** (e.g doing $*$ before $+$) : which, as a language, newOz mostly inherits (or delegates) them from Oz thanks to their shared virtual machine[2].
2. **The parsing/code** : which is the order in which, inside the code, the different expressions and statements alternations are managed.It is quite of really import since the parsing would fail if not ordered properly, but also the performances could take a severe hit.For instance if your most used expression is always at the end of the alternations(given the syntactical precedence is still correct), it might slow the parsing down drastically[3].

3.3.3 Left recursion

Another issue that newOz brings during the parsing is the left recursion. In the context-free grammar, a nonterminal is left-recursive if the leftmost symbol in one of its productions is itself (direct left recursion) or can be made itself by some sequence of substitutions (indirect left recursion). For instance, in newOz unlike in Oz, we can now have the following left recursion problem :

1. When parsing a chained functions call like `f1(p1)(p2)` in newOz with the following **BNF** grammar rules:

<Exp> ::= $\langle Other \rangle$ | $\langle FunCall \rangle$ | ϵ

<FunCall> ::= $\langle Exp \rangle$ params | ϵ

with :A set non-terminal symbols **Exp** (expression), **FunCall** (function call) and **Other** (something else which is can be parsed as an expression). A set of terminals of terminal symbols ϵ and params (function parameters between parenthesis (ex : (1,2)), And the start symbol **Exp**.

---

[2] The complete syntax tree can be found in the appendix). It is to note that the parser won't work correctly without a a perfect syntactic hierarchy and thus still important to remind it.

[3] empirically up to 10 times slower for a bad precedence in the code)

2. We can see that a there is a possible indirect left recursion with **FunCall** and **Exp**. **Such problem wouldn't have happened in Oz** due to its "from out to in" syntax. For instance , our chained function call would be written as `{{F1 P1} P2}` in Oz.

3. Eventually, to solve this issue, we will have a follow a process called left recursion removal [4], and get the result

<Exp> ::= $\langle Other \rangle$
| $\langle FunCall \rangle$
| ϵ

<FunCall> ::= $\langle Other \rangle$ params $\langle FunCallP \rangle$
| $\epsilon params \langle FunCallP \rangle$
|$\langle FunCallP \rangle$

<FunCallP> ::= params $\langle FunCallP \rangle$
| ϵ

This left recursion issue is thus resolvable is another issue to manage in newOz which wasn't in Oz. It appears many times and has a big impact on the code readability (even though a function was created to automatically remove the left recursion from an expression).

3.3.4 Stateless Parsing

The newOz is stateless (doesn't retain any specific or intermediate state during its executing) brings some issues:

- **Assigning a cell outside its declaration(`var x = 4`) is not possible** since the parser won't know whether the named value is a cell or not (so writing `var x; ... x = 4` won't work and won't be accepted). The alternative is to use them the normal way when initialized during their declaration (ex : `x = newCell(4)`). The parser knows many built-in keyword/functions and made them available with both upper and lower cases.

3.4 Reasoning, Tests and Usage

3.4.1 Reasoning

The idea behind this parser is to act as a decorator the the original Oz compiler [5]*Concepts, Techniques, and Models of Computer Programming* June, 2013. Which is why it can also be indirectly called a compiler.

It shall thus grant the access to the same parameters and arguments as the Oz compiler but also enable additional features adapted to newOz.Hence, the newOz compiler is allowed to

---

[4] See how to do it here : https://en.wikipedia.org/wiki/Left_recursion

[5] ozc

do more but not less; for instance it is ok if it fails detecting a issue at parsing time since the oz-compiler behind it will detect it.

3.4.2 Tests

To comply with such goals, the newOz parser/compiler comes with a long list of 200+ unit tests (executed during the git pipeline and while generating releases) meant to detect any newOz parsing issues. They also prioritize detecting false negative (rejecting a parsing when it could be valid in Oz) over false positive (accepting a parsing when it could be not valid in Oz) to allow using the much more refined Oz compiler when there is a doubt.

3.4.3 Usage

The latest release [6](*newOz Parser Releases* June, 2020) and Scala source-code are accessible on a public git repository.

As said before, the executable gives additional options on the top of the ones given(and extended) by the base Oz compiler :

- **-C,–clean** : Clean the intermediary files (default mode).
- **-k,–keep** : Keeps the intermediary ozf files generated during the translation.
- **-f,–folder** : Input and Output Folder to use (current folder if nothing),will compile all the files inside.
- **-F,–force** : Displays newOz compiler message.
- **-v,–verbose** : Displays newOz compiler message.
- **-r,–run**: Runs the first (left-most) newOz file it with the ozengine command.
- **-o FILE,–output=FILE**: Write output to FILE ('-' for stdout).

The overall behaviour of the executable is similar to the one of Oz.

3.5 Examples and results

Here is a list of few examples and results to access the overall performance of the parser and see what it can look like.

---

[6] The release is a jar compatible with any operating system running Java 8+

3.5.1 Switch case

Oz input

```
match elem {
    case p1 => s1
    case p2 => s2
}
```

newOz output

```
case `elem` of `p1` then `s1`
     [] `p2` then `s2`
end
```

3.5.2 Pseudo Currying

newOz , unlike Oz is able to produce a syntax which **visually** (visually only), looks like currying.

Oz input

```

def f2(a, b){
    browse(a,b)
}

def f1(a){
    (b) => {
        f2(a,b)
    }
}

f1("test1")("test2")
```

newOz output

```
fun {`f2``a` `b`}
    {Browse `a` `b`} end

fun {`f1``a`}
    fun {$ `b`}
        {`f2` `a` `b`} end
end


{{`f1` "test1"} "test2"}
```

Longer and more meaning full examples can be found in the Appendix. As for the repository of the parser code and the releases, they can be found here :

- **Public** **newOz** **Parser** Source code link.
- **Public** **newOz** **Parser** Release link.
- Check the ReadMe of the git in case of a of a git repository migration or file hoster change.

Chapter 4

Conclusion

4.1 Programming languages akin to living languages

Section talking about the iterative way the language was created. To support such schema, it is important to write down any idea, thought process, change or observation throughout the whole the whole conception. Personally i used the documentation tool named Confluence [1] to keep track of everything; lest an important idea or detail gets forgotten after a night of sleep resulting in some thought process to be done again [2].

4.2 The difficulties of creating a new Syntax

As adequately conveyed , hopefully, throughout the thesis, creating a new Syntax isn't an undertaking free of troubles.The main issues to face range from taking into account the programming languages "**ecosystem**" to continuously maintaining the overall harmony and consistency of an ongoing solution.

So the main advice I'd give is to keep a clear template complemented guidelines/philosophies as much as possible to avoid turning the problem into a tedious one of testing combinations and permutations just end up redoing all once you stumble upon a big issue.

A good analogy for this would be an AI search tree problem; where you have to prune the possibilities you will probably not want and prioritize some nodes/paths over the others. Confluence is also a good tool to keep track of this kind of things.

4.3 Criticism and weaknesses of the thesis

Let's now try to find the drawbacks and weaknesses of the thesis,to potentially build and improve upon/from them.

---

[1] From Atlassian

[2] The amount of data and ideas stored over there is staggering and would be pleasing to share, but the free version doesn't allow public sharing.

4.3.1 Weakness of definitions

Throughout the whole thesis and project, many definitions are used in either a more or less arbitrary way or with quite a "vague" explanation.
Such were in fact left that way for various reasons with the main ones being:

- **"A popular language"**: for which the definitions and usage stayed on a intuitive level at best. This problem mainly comes that fact that all the the "most popular" programming languages based on different criteria (e.g : interest, salary the language brings, what is used in the industries etc..)
- **"A modern language"**: while this definition is more explicit, a part of it relies on the definition of **"A popular language"** and thus also inherits the same issues.
- **"Teaching issues"**: the teaching/learning issues related to Oz cited are not backed by empirical studies but only by intuitions/observations or following the status-quo.
- **"More/less pleasing aesthetics"**: can't easily or shortly define what makes a language (programming or not) more pleasing than another aside from doing a raised hand vote during debates.

4.3.2 Missed opportunities

Like any relatively constrained project (in term of resources), some things would have been really nice to have but weren't that high on priority.

1. **newOz could have been more free in term of expressions/statements ordering**:newOz now allows the user to write some repeated expression indiscriminately of their order. Take newOz's **Functors** or **Classes and Objects** for example; we can now interleave imports/-export, values/variable or attributes/properties when declaring them as we would in many modern languages. One of the greatly missed opportunity was to not extend such schema everywhere it could have been possible (e.x : meths interleaved with values/variable). The motivations being this omission mainly comes from the lack of perspective on newOz; such changes couldn't be done so easily everywhere without many trials and debates.Thus the risk wasn't taken when not needed. What we have in term of ordering is already quite pleasing when using newOz
2. **newOz could have been fully made on a new low level compiler + parser**: but wasn't since it could potentially takes twice as much time as the current solution (newOz to Oz parser).
3. **newOz could have benefited from some kind of field test with random users** : but i didn't mainly due to how young it was and how much time is needed for that (e.g : finding an user base and aggregating their feedback).
4. **newOz Emacs interactive mod**: whether it is due to time limitation to create the mod itself or allowing partial parsing (the current parser only supports end to end parsing , stopping mid way isn't possible), the interactivity of Oz couldn't have been brought physically to newOz.

5. **Implementing a newOz to Oz parser** (for reverse parsing): Adapting old Oz codes into newOz manually isn't really feasible, the best way would be to have a have a reverse parser; While some ground-work was done in that direction (some early Oz to newOzimplementation) , it was evaluated that finishing and making such parser as robust as the newOz one would be too consuming. If you have time and resources to work a lot on retro-compatibility, you could also use that time to work on something more useful for newOz (like the **Mozart** mod for interactive programming).

6. **Method names** was one (if not the only one) syntactical choice trully taken with technical and historical limitation in mind (current newOz stateless parser + Oz way to differentiate private and public). So even though the current newOz formulation is new, it already inherits of some form of historical limitation which should be avoided when possible[3].

4.4 Conclusion

So here we are, almost at the end of this short but packed journey ...

newOz, from its inception to its realisation, has (aside from a few but important points listed in the following sections) , for me, truly reached its goal of being a "New Syntax for Oz".
What is brought to Oz is the following:

1. It brought back on the table (and tried to treat them) some interesting subjects of Oz (e.x : feats in classes, upper/lower cases, function calls etc..).

2. A much more familiar (as of today) programming syntax.

3. A new path to thread on (see next section).

4. While writing a lot of tests and code examples, it felt quite good.

4.5 The future of the new syntax

In closing,we can also touch up on the subject of "what's next".

The future of the new syntax is highly related to also solving its main flaw which is maturation. It shall be used as a references and improved upon to hopefully be used as teaching material. So the real next Oz might be newOz or something based on it but newOz is definitely a huge step toward the right direction.
As said before, in the Appendinx, you will find:

- The newOz **BNF** grammar in juxtaposed to the one of Oz [4].

- A few concrete and executable code examples in newOz, Oz and the generated Oz via the newOz parser/compiler.

Thanks a lot for reading this thesis.

---

[3] (or else users will often come to you asking "**WHY ?**")

[4] The Scala syntax summary can be found here: .

REFERENCES

"A History of the Oz Multiparadigm Language" (n.d.). In: *Proc. ACM Program. Lang.* 4.1 (), p. 56. DOI: 10.1145/3386333. URL: https://doi.org/10.1145/3386333.

Compiler Wiki Page (n.d.). URL: https://en.wikipedia.org/wiki/Compiler.

Concepts, Techniques, and Models of Computer Programming (June, 2013). The MIT press. ISBN: 978-0262220699.

Doeraene, Sébastien (2003). "Ozma: Extending Scala with Oz Concurrency". MA thesis.

Elixir Wiki Page (n.d.). URL: https://en.wikipedia.org/wiki/Elixir_(programming_language).

Lisp variable official doc (June, 2020). URL: https://www.tutorialspoint.com/lisp/lisp_variables.htm.

newOz Parser Releases (June, 2020). URL: https://bitbucket.org/JeanGaka/oz-new-syntax-parser/downloads/?tab=downloads.

Parsers Wiki Page (n.d.). URL: https://en.wikipedia.org/wiki/Parsing.

Scala docs (n.d.). URL: https://docs.scala-lang.org/.

Scala Parser and Combinators doc (n.d.). URL: https://www.scala-lang.org/api/2.12.3/scala-parser-combinators/scala/util/parsing/combinator/Parsers.html.

Scala Wiki Page (n.d.). URL: https://en.wikipedia.org/wiki/Scala_(programming_language).

Seif Haridi, Nils Franzen (2013). *Tutorial of Oz*. URL: http://mozart2.org/mozart-v1/doc-1.4.0/tutorial/index.html.

Appendix A

Appendix C

C.1 Oz Original Syntax(nested)

<interStatement> ::= ⟨*statement*⟩
| ‘`declare`’ { ⟨*declarationPart*⟩ }+ [⟨*interStatement*⟩]
| ‘`declare`’ { ⟨*declarationPart*⟩ }+ ‘`in`’ ⟨*interStatement*⟩

Table C.1: Interactive statements (*Concepts, Techniques, and Models of Computer Programming* June, 2013)

<statement> ::= ⟨*nestCon(statement)*⟩ |⟨*nestDec(variable)*⟩
| ‘`skip`’ | ⟨*statement*⟩ ⟨*statement*⟩

<expression> ::= ⟨*nestCon(expression)*⟩ |⟨*nestDec(‘$’)*⟩
| ⟨*expression*⟩ ⟨*evalBinOp*⟩ ⟨*expression*⟩
| ‘$’ | term | ‘’ ⟨*expression*⟩ | ‘`self`’

<inStatement> ::= [{ ⟨*declarationPart*⟩ }+ ‘`in`’]⟨*statement*⟩

<inExpression> ::= [{ ⟨*declarationPart*⟩ }+ ‘`in`’][⟨*statement*⟩]⟨*expression*⟩

<in(statement)> ::= ⟨*inStatement*⟩

<in(expression)> ::= ⟨*inExpression*⟩

Table C.2: Statements and expressions (*Concepts, Techniques, and Models of Computer Programming* June, 2013)

The operators/keyword are alongside newOz’s in the next section to compare their changes. The precedence stay the same.

C.2 newOz Syntax(nested)

This section will reuse the terms and formats used previously with the main addition of the parameter separator "," which is meant to be used between a list of parameter (whether it is for a table or a function call for instance). The comma won’t be writing like it should (only accepting it between two idioms) to keep everything overall readable.

```
    <nestCon(α)> ::= ⟨expression⟩ ( ‘=’ | ‘:=’ | ‘,’ ) ⟨expression⟩
|   ‘{’ ⟨expression⟩{⟨expression⟩} ‘}’
|   local {⟨declarationPart⟩} + in [⟨statement⟩ ]⟨α⟩ end
|   ‘(’ ⟨in(α)⟩ ‘)’
|   if ⟨expression⟩ then ⟨in(α)⟩
    { elseif ⟨expression⟩ then ⟨in(α)⟩}
    [else ⟨in(α)⟩ ]end
|   case ⟨expression⟩ of ⟨pattern⟩ [andthen ⟨expression⟩ ]then ⟨in(α)⟩
    { ‘[]’ ⟨pattern⟩ [andthen ⟨expression⟩ ]then ⟨in(α)⟩}
    [else ⟨in(α)⟩ ]end
|   for {⟨loopDec⟩} + do ⟨in(α)⟩ end
|   try ⟨in(α)⟩
    [catch ⟨pattern⟩ then ⟨in(α)⟩
    { ‘[]’ ⟨pattern⟩ then ⟨in(α)⟩} ]
    [finally ⟨in(α)⟩ ]end
|   raise ⟨inExpression⟩ end
|   thread ⟨in(α)⟩ end
|   lock [⟨expression⟩ then ]⟨in(α)⟩ end
```

Table C.3: Nestable constructs (no declarations) (*Concepts, Techniques, and Models of Computer Programming* June, 2013)

```
    <nestDec(α)> ::= proc ‘{’ α {⟨pattern⟩} ‘}’ ⟨inStatement⟩ end
|   fun [lazy ]‘{’ α {⟨pattern⟩} ‘}’ ⟨inExpression⟩ end
|   functor α
    [import {⟨variable⟩ [at ⟨atom⟩ ]
|   ⟨variable⟩ ‘(’
    { (⟨atom⟩|⟨int⟩)[‘:’ ⟨variable⟩ ]}+ ‘)’
    }+]
    [export { [( ⟨atom⟩| ⟨int⟩) ‘:’ ]⟨variable⟩} +]
    define {⟨declarationPart⟩} +[in ⟨statement⟩ ]end
|   class α {⟨classDescriptor⟩}
    { meth ⟨methHead⟩ [‘=’ ⟨variable⟩ ]
    ( ⟨inExpression⟩| ⟨inStatement⟩ ) end }
    end
```

Table C.4: Nestable declarations (*Concepts, Techniques, and Models of Computer Programming* June, 2013)

```
    <term> ::= [‘!’ ]⟨variable⟩| ⟨int⟩|⟨float⟩|h c⟨aracter⟩
|   ⟨atom⟩|⟨string⟩| unit | true | false
|   ⟨label⟩ ‘(’ { [⟨feature⟩ ‘:’ ]⟨expression⟩} ‘)’
|   ⟨expression⟩⟨consBinOp⟩⟨expression⟩
|   ‘[’ {⟨expression⟩} + ‘]’

    <pattern> ::= [‘!’ ]⟨variable⟩| ⟨int⟩|⟨float⟩| ⟨character⟩
|   ⟨atom⟩|⟨string⟩| unit | true | false
|   ⟨label⟩ ‘(’ { [⟨feature⟩ ‘:’ ]⟨pattern⟩} [‘...’ ]‘)’
|   ⟨pattern⟩⟨consBinOp⟩⟨pattern⟩
|   ‘[’ {⟨pattern⟩} + ‘]’
```

Table C.5: Terms and patterns (*Concepts, Techniques, and Models of Computer Programming* June, 2013)

```
    <declarationPart> ::= ⟨variable⟩|⟨pattern⟩ ‘=’ ⟨expression⟩| ⟨statement⟩

    <loopDec> ::= ⟨variable⟩ in ⟨expression⟩ [‘..’ ⟨expression⟩ ][‘;’ ⟨expression⟩ ]
|   ⟨variable⟩ in ⟨expression⟩ ‘;’ ⟨expression⟩ ‘;’ ⟨expression⟩
|   break ‘:’ ⟨variable⟩| continue ‘:’ ⟨variable⟩
|   return ‘:’ ⟨variable⟩| default ‘:’ ⟨expression⟩
|   collect ‘:’ ⟨variable⟩

    <binaryOp> ::= ⟨evalBinOp⟩|⟨consBinOp⟩

    <consBinOp> ::= ‘#’ | ‘|’

  <evalBinOp> ::= ‘+’ | ‘−’ | ‘*’ | ‘/’ | div | mod | ‘.’ | andthen
| orelse | ‘:=’ | ‘,’ | ‘=’ | ‘==’ | ‘’⁻′|‘<’|‘=<’|‘>’|‘>=’

    <label> ::= unit | true | false |⟨variable⟩| ⟨atom⟩

    <feature> ::= unit | true | false |⟨variable⟩| ⟨atom⟩| ⟨int⟩

    <classDescriptor> ::= from {⟨expression⟩} + | prop {⟨expression⟩} +
|   attr {⟨attrInit⟩} +

    <attrInit> ::= ( [‘!’ ]⟨variable⟩|⟨atom⟩| unit | true | false )
    [‘:’ ⟨expression⟩ ]

    <methHead> ::= ( [‘!’ ]⟨variable⟩|⟨atom⟩| unit | true | false )
    [‘(’ {⟨methArg⟩} [‘...’ ]‘)’ ]
    [‘=’ ⟨variable⟩ ]

    <methArg> ::= [⟨feature⟩ ‘:’ ]( ⟨variable⟩| ‘_’ | ‘$’ )[‘<=’ ⟨expression⟩ ]
```

Table C.6: Other nonterminals needed for statements and expressions (*Concepts, Techniques, and Models of Computer Programming* June, 2013)

<variable> ::= (uppercase char) { (alphanumeric char) }
| ’‘’ {⟨*variableChar*⟩|⟨*pseudoChar*⟩} ’‘’

<atom> ::= (lowercase char) { (alphanumeric char) } (except no keyword)
| ”’ {⟨*atomChar*⟩ |⟨*pseudoChar*⟩ } ”’

<string> ::= ‘`"`’ {⟨*stringChar*⟩ | ⟨*pseudoChar*⟩} ‘`"`’

<character> ::= (any integer in the range 0 ... 255)
| ‘`&`’ ⟨*charChar*⟩| ‘`&`’ ⟨*pseudoChar*⟩

Table C.7: Lexical syntax of variables, atoms, strings, and characters (*Concepts, Techniques, and Models of Computer Programming* June, 2013)

<interStatement> ::= ⟨*statement*⟩
| ‘`declare`’ ‘`{`’ { ⟨*declarationPart*⟩}+ [⟨*interStatement*⟩]‘`}`’

Table C.8: Interactive statements.

<statement> ::= ⟨*nestCon(statement)*⟩ |⟨*nestDec(variable)*⟩
| ‘`skip`’ | ⟨*statement*⟩ ⟨*statement*⟩

<expression> ::= ⟨*nestCon(expression)*⟩ |⟨*nestDec(‘$’)*⟩
| ⟨*expression*⟩ ⟨*evalBinOp*⟩ ⟨*expression*⟩
| ‘$’ | term | ‘’ ⟨*expression*⟩ | ‘`this`’

<inStatement> ::= ‘`{`’ [{ ⟨*declarationPart*⟩}+]⟨*statement*⟩ ‘`}`’

<inExpression> ::= ‘`{`’[{ ⟨*declarationPart*⟩}+][⟨*statement*⟩]⟨*expression*⟩‘`}`’

<in(statement)> ::= ⟨*inStatement*⟩

<in(expression)> ::= ⟨*inExpression*⟩

<blockIn(α)> := ‘`{`’(⟨*in(α)*⟩ | ‘ ’)‘`}`’

Table C.9: Statements and expressions.

```
  <nestCon(α)> ::= ⟨expression⟩ ( ‘=’ | ‘:=’ ) ⟨expression⟩
| ⟨expression⟩‘(’ {⟨expression⟩ ‘,’ } ‘)’
| ⟨blockIn(α)⟩
| ‘(’ ⟨in(α)⟩ ‘)’
| if ‘(’⟨expression⟩‘)’ ⟨blockIn(α)⟩
  { else if ‘(’⟨expression⟩‘)’ ⟨blockIn(α)⟩’}
  [else ⟨blockIn(α)⟩ ]
| match ⟨expression⟩ ‘{’
  { ‘case’ ⟨pattern⟩ [(‘&&’ | ‘||’) ⟨expression⟩ ]‘=>’ (⟨blockIn(α)⟩ | ⟨in(α)⟩) }+
  [else ⟨blockIn(α)⟩ ]‘}’
| for ‘(’{⟨loopDec⟩} + ‘)’⟨blockIn(α)⟩
| try ⟨blockIn(α)⟩
  [catch ‘{’ { case ⟨pattern⟩ ‘=>’ (⟨blockIn(α)⟩ | ⟨in(α)⟩) } ‘}’ ]
  [finally ⟨blockIn(α)⟩ ]
| raise ⟨blockIn(expression)⟩
| thread ⟨blockIn(α)⟩
| lock [‘(’⟨expression⟩ ‘)’ ]⟨blockIn(α)⟩
```

Table C.10: Nestable constructs (no declarations).

```
  <nestDec(α)> ::= defproc α‘(’ {⟨pattern⟩ ‘,’ } ‘)’ ⟨blockIn(statement)⟩
| def [lazy ]α‘(’ {⟨pattern⟩ ‘,’ } ‘)’ ⟨blockIn(expression)⟩
| functor α? {
  [(import ({⟨variable⟩ [at ⟨atom⟩ ]| ⟨variable⟩ ‘(’ { (⟨atom⟩|⟨int⟩)[‘:’ ⟨variable⟩ ]‘,’ }+
  ‘)’ ) ‘,’ }+]
  [export { [( ⟨atom⟩| ⟨int⟩) ‘:’ ]⟨variable⟩ ‘,’ } +]}*
  ⟨blockIn(statement)⟩
| class α {⟨classDescriptor⟩} ‘{’
  { def ⟨methHead⟩ [‘=’ ⟨variable⟩ ]
  ( ⟨blockIn(expression)⟩ | ⟨blockIn(statement)⟩ ) }
  ‘}’
```

Table C.11: Nestable declarations.

```
    <term> ::= [‘!’ ]⟨variable⟩| ⟨int⟩|⟨float⟩|h c⟨aracter⟩
 |  ⟨atom⟩|⟨string⟩| unit | true | false
 |  ⟨label⟩ ‘(’ { [⟨feature⟩ ‘:’ ]⟨expression⟩ ‘,’ } ‘)’
 |  ⟨expression⟩⟨consBinOp⟩⟨expression⟩
 |  ‘[’ {⟨expression⟩ ‘,’} + ‘]’

    <pattern> ::= [‘!’ ]⟨variable⟩| ⟨int⟩|⟨float⟩| ⟨character⟩
 |  ⟨atom⟩|⟨string⟩| unit | true | false
 |  ⟨label⟩ ‘(’ { [⟨feature⟩ ‘:’ ]⟨pattern⟩ ‘,’ } [‘,’ ‘...’ ]‘)’
 |  ⟨pattern⟩⟨consBinOp⟩⟨pattern⟩
 |  ‘[’ {⟨pattern⟩} + ‘]’
```

Table C.12: Terms and patterns.

```
    <declarationPart> ::= {(‘val’ | ‘var’ ) {(⟨variable⟩|⟨pattern⟩) ‘=’ (⟨expression⟩|
    ⟨statement⟩) ‘,’}+ ‘;’? }

    <loopDec> ::= ⟨variable⟩ in ⟨expression⟩ [‘..’ ⟨expression⟩ ][‘;’ ⟨expression⟩ ]
 |  ⟨variable⟩ in ⟨expression⟩ ‘;’ ⟨expression⟩ ‘;’ ⟨expression⟩
 |  break ‘:’ ⟨variable⟩| continue ‘:’ ⟨variable⟩
 |  return ‘:’ ⟨variable⟩| default ‘:’ ⟨expression⟩
 |  collect ‘:’ ⟨variable⟩

    <binaryOp> ::= ⟨evalBinOp⟩|⟨consBinOp⟩

    <consBinOp> ::= ‘#’ | ‘|’

  <evalBinOp> ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’ | ‘.’
| ‘&&’ | ‘||’ | ‘:=’ | ‘,’ | ‘=’ | ‘==’ | ‘’−′|‘<’|‘=<’|‘>’|‘>=’

    <label> ::= unit | true | false |⟨variable⟩| ⟨atom⟩

    <feature> ::= unit | true | false |⟨variable⟩| ⟨atom⟩| ⟨int⟩

    <classDescriptor> ::= extends {⟨expression⟩} + | prop {⟨expression⟩} +
 |  attr {⟨attrInit⟩} +

    <attrInit> ::= ( [‘!’ ]⟨variable⟩|⟨atom⟩| unit | true | false )
    [‘:’ ⟨expression⟩ ]

    <methHead> ::= ( [‘!’ ]⟨variableStric⟩|⟨atomLisp⟩| unit | true | false )
    [‘(’ {⟨methArg⟩ ‘,’ } [‘,’ ‘...’ ]‘)’ ]
    [‘=’ ⟨variable⟩ ]

    <methArg> ::= [⟨feature⟩ ‘:’ ]( ⟨variable⟩| ‘_’ | ‘$’ )[‘=’ ⟨expression⟩ ]
```

Table C.13: Other non-terminals needed for statements and expressions.

<variableStrict> ::= (upper case char) { (alphanumeric char) }
| ’‘’ {⟨*variableChar*⟩|⟨*pseudoChar*⟩} ’‘’

<variable> ::= (upper or lower case char) { (alphanumeric char) }
| ’‘’ {⟨*variableChar*⟩|⟨*pseudoChar*⟩} ’‘’

<atom> ::= ⟨*atomLisp*⟩
| ”’ {⟨*atomChar*⟩ |⟨*pseudoChar*⟩ } ”’

<string> ::= ‘"’ {⟨*stringChar*⟩ | ⟨*pseudoChar*⟩} ‘"’

<character> ::= (any integer in the range 0 ... 255)
| ‘&’ ⟨*charChar*⟩| ‘&’ ⟨*pseudoChar*⟩

<atomLisp> ::= ‘’ (lower or upper case char) { (alphanumeric char) } (except no keyword)

Table C.14: Lexical syntax of variables, atoms, strings, and characters.

On a side note;The ternary operator ". :=" vs the binary operator ":=" are still managed as they were in Oz (see *Concepts, Techniques, and Models of Computer Programming* June, 2013) . Secondly the rest of the definitions (aside from the keyword) are the same as they were in Oz (what is a char, an int etc..) .

C.3 Keywords changes

andthen =>&&	default	**false**	**lock**	return
at	**define =>removed**	**feat(*)**	**meth**	**self =>this**
attr	**dis(*)**	**finally**	**mod**	**skip**
break	**div =>/**	**for**	**not(*)**	**then =>removed (replaced by =>generaly)**
case =>match	**do**	**from**	**of**	**thread**
catch	**else**	**fun =>def**	**or(*)**	**true**
choice	**elsecase(*)**	**functor**	**orelse**	**try**
class	**elseif =>else if**	**if**	**prepare(*)**	**unit**
collect	**elseof(*)**	**import**	**proc =>defproc**	4*
cond(*)	**end =>removed (replaced by } generaly)**	**in**	**prop**	
continue	**export**	lazy	**raise**	
declare	**fail**	**local =>removed (replaced by { generaly)**	**require(*)**	

Table C.15: Oz Keywords alongside their changes in newOz (if changed). [1]

Appendix E

E.1 Examples of code and parsings

Every code example [1] (whether Oz or newOz can be parsed and compiled (even tho the result might be useless)).

This chapter is structured as following (for each example) :

1. An Oz code examples.
2. Its newOz handwritten as i would write them in a day to day usage.
3. ThenewOz → Oz translated code generated by the parser release (using the "–keep" arg to show the files) [2]. The code is untouched so any formatting discrepancy is due to the parser's limitations.

Additionally , after installing the required binaries :

- The Oz code examples shown here can be compiled with "**ozc -c <filePath>** " and ran with "**ozengine <ozfFilePath>** ".
- The newOz code can be build and ran at the same time with the command "**java -jar <releaseJarPath> -c -r –keep <filePath>**"[3] .
- The newOz examples contain some comments for additional syntax related information.
- All of them were complied easily and quickly under the current release (v1,2).

---

[1] All the code example shown here were taken/adapted from (*Concepts, Techniques, and Models of Computer Programming* June, 2013)

[2] Released used at the day of the Thesis submission, thus v1.2

[3] A release can easily be generating directly from the source available in the repository with the command"sbt assembly" if needed (release versions are tagged and follow a test pipeline)

E.1.1 Ball Game

Oz

```
functor
import Browser(browse:Browse)
define B1 B2 NewActive BallGame in
  fun {NewActive Class Init}
    Obj={New Class Init}
    P
  in
    thread S in
      {NewPort S P}
      for M in S do {Obj M} end
    end
    proc {$ M} {Send P M} end
  end
  class BallGame
    attr other count:0

    meth init(Other)
      other:=Other
    end
    meth ball
      count:=@count+1
      {@other ball}
    end
    meth get(X)
      X=@count
    end
  end
  B1={NewActive BallGame init(B2)}
  B2={NewActive BallGame init(B1)}
  {B1 ball}
  local X in {B1 get(X)} {Browse X} end
end
```

Listing E.1: "BallGame.oz Oz version"

newOz

```
functor
import Browser('browse:browse)
{
  val b1,b2,newActive,BallGame;
  // Writting Class since the lower case version is a keyword
  def newActive(Class,init){
    val obj= new(Class,init)
    val p
    thread{
      val s
      newPort(s,p)
      for(m in s){ obj(m)}
    }
    (m){ send(p,m) }
  }
  class BallGame{
    attr other
    attr count:0

    def 'init(Other){
      other:=Other
    }
    def 'ball(){
      count:=@count+1
      @other('ball)
    }
    def 'get(x){
      x=@count
    }
  }
  b1 = newActive(BallGame,'init(b2))
  b2 = newActive(BallGame,'init(b1))
  b1('ball)
  {
      val x;
      b1('get(x))
      browse(x)
  }
}
```

Listing E.2: "BallGame.noz Oz version"

newOz compiled to Oz

```
functor
import Browser(browse:`browse`)
define
   `b1` `b2` `newActive` BallGame in
  fun {`newActive` Class `init`}
     `obj`= {New Class `init`}
       `p` in
    thread
       `s` in
      {NewPort `s` `p`}
      for `m` in `s` do {`obj` `m`} end
      end
    proc {$ `m`} {Send `p` `m`} end
   end
  class BallGame
    attr `other` `count`:0

    meth init(Other)
      `other`:=Other
     end
    meth ball
      `count`:=@`count`+1
      {@`other` ball}
     end
    meth get(`x`)
      `x`=@`count`
     end
  end
  `b1` = {`newActive` BallGame init(`b2`)}
   `b2` = {`newActive` BallGame init(`b1`)}
   {`b1` ball}

      local `x` in
      {`b1` get(`x`)}
      {`browse` `x`}
   end

end
```

Listing E.3: "BallGame.oz compiled newOz

E.1.2 Fibonacci

Oz

```
functor
import Browser(browse:Browse)
define C Fibo NewStat ComputeServer in
  fun {NewStat Class Init}
  P Obj={New Class Init} in
    thread S in
       {NewPort S P}
       for M#X in S do
         try {Obj M} X=normal
         catch E then X=exception(E) end
       end
    end
    proc {$ M}
    X in
       {Send P M#X}
       case X of normal then skip
       [] exception(E) then raise E end end
    end
  end
  class ComputeServer
    meth init skip end
    meth run(P) {P} end
  end
  C={NewStat ComputeServer init}
  fun {Fibo N}
    if  N<2 then 1 else {Fibo N-1}+{Fibo N-2} end
  end
  local F in
    {C run(proc {$} F={Fibo 30} end)}
    {Browse F}
  end
  local F in
    F={Fibo 30}
    {Browse F}
  end
end
```

Listing E.4: "Fibo.oz Oz version"

newOz

```
functor
import Browser('browse:browse)
{
  val c,fibo ,newStat ,ComputeServer;
  def newStat(Class ,init){
        val p,obj= new(Class ,init);
        thread{
            val s;
            newPort(s,p)
            for(m#x in s){
                try{
                    obj(m)
                    x='normal
                }catch{
                    case e => x='exception(e)
                }
            }
        }
        defproc $(m){
            val x;
            send(p,m#x)
            match x {
                case 'normal => skip
                case 'exception(e) => raise{e}
            }
        }
  }
  class ComputeServer{
      def 'init(){skip}
      def 'run(p){p()}
    }
    c=newStat(ComputeServer ,'init)
    def fibo(n){
        if (n<2) { 1 }else{fibo(n-1) + fibo(n-2)}
    }
    {
        val f;
        c('run((){f=fibo(30)}))
        browse(f)
    }
    {
        val f;
        f=fibo(30)
        browse(f)
    }
}
```

Listing E.5: "Fibo.noz Oz version"

newOz compiled to Oz

```
functor
import Browser(browse:`browse`)
define
   `c`  `fibo`  `newStat`  ComputeServer  in
  fun {`newStat` Class `init`}
         `p`  `obj`= {New Class `init`}  in
        thread
             `s`  in
            {NewPort `s` `p`}
            for `m`#`x` in `s` do
                try
                    {`obj` `m`}
                    `x`=normal
                 catch `e` then `x`=exception(`e`)
                   end
              end
          end
        proc {$  `m`}
             `x`  in
            {Send `p` `m`#`x`}
            case `x`
                of normal then skip
                 [] exception(`e`) then raise `e` end
            end
          end
   end
  class ComputeServer
      meth init skip end
      meth run(`p`) {`p` } end
    end
    `c`={`newStat` ComputeServer init}
     fun {`fibo` `n`}
        if  `n`<2  then  1 else {`fibo` `n`-1} + {`fibo` `n`-2}  end
     end

        local  `f`  in
        {`c` run(proc {$ } `f`={`fibo` 30}  end)}
        {`browse` `f`}
     end

        local  `f`  in
        `f`={`fibo` 30}
        {`browse` `f`}
     end

end
```

Listing E.6: "Fibo.oz compiled newOz

E.1.3 Flavius Josephus

Oz

```
%% The Flavius Josephus problem (active object version)
functor
import Browser(browse:Browse)
define NewActive Victim Josephus in
  class Victim
    attr ident step last succ pred alive:true
    meth init(I K L) ident:=I step:=K last:=L end
    meth setSucc(S) succ:=S end
    meth setPred(P) pred:=P end
    meth kill(X S)
      if @alive then
      if S==1 then @last=@ident
      elseif X mod @step==0 then
        alive:=false
        {@pred newsucc(@succ)}
        {@succ newpred(@pred)}
        {@succ kill(X+1 S-1)}
      else
        {@succ kill(X+1 S)}
      end
      else {@succ kill(X S)} end
    end
    meth newsucc(S)
      if @alive then succ:=S
      else {@pred newsucc(S)} end
    end
    meth newpred(P)
      if @alive then pred:=P
      else {@succ newpred(P)} end
    end
  end

  fun {Josephus N K}
    A={NewArray 1 N null}
    Last
  in
    for I in 1..N do
      A.I:={NewActive Victim init(I K Last)}
    end
    for I in 2..N do {A.I setPred(A.(I-1))} end
    {A.1 setPred(A.N)}
    for I in 1..(N-1) do {A.I setSucc(A.(I+1))} end
    {A.N setSucc(A.1)} {A.1 kill(1 N)}
    Last
  end
end
```

Listing E.7: "Josephus.oz Oz version"

newOz

```
// The Flavius Josephus problem (active object version)
functor
import Browser('browse:browse)
{
    // Showing another way to declare
  val newActive,Victim
  val josephus
  class Victim{
    attr ident,step, last;  // attributes can be writing like you would for
    vals
    attr succ,pred, alive:true // with or without ; at the end
    def 'init(i,k,l) {ident:=i step:=k last:=l }
    def 'setSucc(S) {succ:=S }
    def 'setPred(P) {pred:=P }
    def 'kill(X,S){
      if(@alive){
      if(S==1){ @last=@ident}
      else if(X % @step==0){
        alive:=false
        @pred('newsucc(@succ))
        @succ('newpred(@pred))
        @succ('kill(X+1,S-1))
      }else{
        @succ('kill(X+1,S))
      }
      }else{ @succ('kill(X,S)) }
    }
    def 'newsucc(S){
      if(@alive) {succ:=S}
      else {@pred('newsucc(S)) }
    }
    def 'newpred(P){
      if(@alive){pred:=P}
      else {@succ('newpred(P)) }
    }
  }

  def josephus(n,k){
    /*Using upper case to write NewArray since there is no warrenty
    (for this example let's assume it's not) that the newOz parser has it
   in its lookup table
    */
    val a=NewArray(1,n,null)
    val last
    for(i in 1..n){
      a.i:= newActive(Victim,'init(i,k,last)) // Here newActive is used in
   a newOz way since known by the parser
    }
    // a lil more complex since the labelcall in Oz looks like a function
   call in newOz
    for(i in 2..n){a.i('setPred(a.(i-1)))}
    a.1('setPred(a.n))
    for(i in 1..(n-1)){ a.i('setSucc(a.(i+1)))}
```

```
    a.n('setSucc(a.1)) a.1('kill(1,n))
    last
  }
  browse(josephus(12,10))
}
```

Listing E.8: "Josephus.noz Oz version"

newOz compiled to Oz

```
%% The Flavius Josephus problem (active object version)
 functor
import Browser(browse:`browse`)
define
    %% Showing another way to declare
    `newActive`  Victim  `josephus`  in
    %% Showing another way to declare
  class Victim
    attr `ident` `step`  `last`  `succ` `pred`  `alive`:true %% with or
   without ; at the end
    meth init(`i` `k` `l`)  `ident`:=`i` `step`:=`k`  `last`:=`l`  end
    meth setSucc(S)  `succ`:=S  end
    meth setPred(P)  `pred`:=P  end
    meth kill(X S)
      if @`alive`  then
      if S==1  then @`last`=@`ident`
      elseif X mod @`step`==0  then
        `alive`:=false
        {@`pred` newsucc(@`succ`)}
         {@`succ` newpred(@`pred`)}
         {@`succ` kill(X+1 S-1)}
       else
        {@`succ` kill(X+1 S)}
        end
       else {@`succ` kill(X S)}  end
     end
    meth newsucc(S)
      if @`alive`  then  `succ`:=S
      else  {@`pred` newsucc(S)}  end
     end
    meth newpred(P)
      if @`alive`  then `pred`:=P
      else  {@`succ` newpred(P)}  end
     end
  end

  fun {`josephus` `n` `k`}
    /*Using upper case to write NewArray since there is no warrenty
    (for this example let's assume it's not) that the newOz parser has it
   in its lookup table
    */
     `a`={NewArray 1 `n` null}
       `last` in
    /*Using upper case to write NewArray since there is no warrenty
    (for this example let's assume it's not) that the newOz parser has it
   in its lookup table
    */
    for `i` in 1..`n` do
      `a`.`i`:= {`newActive` Victim init(`i` `k` `last`)} %% Here newActive
    is used in a newOz way since known by the parser
      end
    %% a lil more complex since the labelcall in Oz looks like a function
   call in newOz
```

```
    for `i` in 2..`n` do {`a`.`i` setPred(`a`.(`i`-1))}  end
    {`a`.1 setPred(`a`.`n`)}
     for `i` in 1..(`n`-1) do {`a`.`i` setSucc(`a`.(`i`+1))}  end
     {`a`.`n` setSucc(`a`.1)}  {`a`.1 kill(1 `n`)}
    `last`
   end
  {`browse` {`josephus` 12 10}}

end
```

Listing E.9: "Josephus.oz compiled newOz

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**