A review of Oz and its implementation with Mozart

Philippe Giabbanelli
Bishop's University, Undergraduate 'Principles of Programming Languages' Course
aqualonne@free.fr

Abstract

Oz is a modern programming language aiming at unifying several paradigms: object-oriented programming, functional programming, logic and constraint programming, with concurrency and distribution of computations over a network. We will study how it tries in its third version to satisfy all these paradigms into one design, answering the basic questions of syntax, data type and scopes; some aspects of the language will be studied further in details such as the object oriented. Finally, we discuss the applications of Oz so far and its evolution, mentioning one of its successors: Alice.

1. Overview of the evolution

The development of Oz started in 1991 at DFKI[1] under the lead of Gert Smolka[2]. The goal was to advance ideas from constraint and concurrent logic programming and to develop a pratically useful programming system. The group arrived in 1994 at the final base language and a stable implementation (FDKI Oz). In 1996, they founded the Mozart Consortium with SICS and Louvain-la-Neuve. Mozart 1.0 was released in January 1999 [1]. The development continued with an international group, the Mozart Consortium, which consisted mainly of Universität des Saarlandes, the Swedish Institute of Computer Science (SICS), and the Université catholique de Louvain. In 2005, the responsibility for managing Mozart development was transferred to the Mozart Board[3].

Oz is still implemented by the Mozart System. The original Oz 1 computation model, supported a fine-grained notion of concurrency where each statement could be executed concurrently; the experience showed that it was hard to debug and awkward in some aspects. Oz 2 uses instead a thread-based concurrency model with explicit creation of threads. Oz 3 extends Oz 2 with support for incremental construction of programs and synchronization over the internet [3].

[1] Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), *lit.* German Research Center for Artificial Intelligence, the largest AI research center in the world.
[2] Head of Programming Systems lab at DFKI in 1990-1998
[3] The more actives members are Peter Van Roy, Gert Smolka and Seif Haridi, all computer scientists. For developments in Linguistic, see Torbjörn Lager.

2. Main principles of the language

1) Why a multi-paradigm language ?

A good example is given by Peter Van Roy when he teaches the Concepts of Programming Languages. One would like to study several paradigms in order to broaden his culture, but he has to face many languages such as Java for object oriented, Haskell for functional, Erlang for concurrency, Prolog, etc. Studying a new language for each new paradigm implies a change of syntax, a change of semantic and of environment. Thus, being able to keep the same language for several paradigms is quite an important gain of time. This works, in theory, as well in the industry : when a language shows the limitation of its paradigm, one may have to use another language and interface them; if we could stay within the same language, we could gain productivity. In fact, having multi-paradigm sounds in itself like a very good idea that everybody should have : the problem is that one cannot just merge all paradigms. As we will see, the main point is to have a single coherent design.

2) What Oz has from each paradigm

· Object Oriented Programming : state, abstract data types, classes, objets, (multiple) inheritance.
· Functional Programming : every entity is first class, lexical scoping[4], small Kernel (*c.f.* syntax).
· Logic and constraint programming : logic variables, disjunctive constructs, programmable search strategies.
· Concurrent : dynamic creation of (dataflow) threads, interacting with each other.

[4] *i.e.* static scope, determined when the code is compiled. Variable are only visible within the block of the definition.

3) Where does Oz come from ?

Nowadays, it would be surprising that a new language comes out without any influence. In the case of Oz, the design and implementation took ideas from AKL [5,6], which stands for Andorra Kernel Language. It is a concurrent constraint language with encapsulated search, developed in the 90, with the programming paradigms of both Prolog and GHC; there are means to structure search in a 'better' way than the original backtracking of Prolog. AKL is based on an instance of KAP (Kernel Andorra Prolog [7]), developed by the same authors.

KAP is defined as a logic programming language based on constraints; it is designed so that Prolog, GHC, Parlog[5] and Atomic Herbrand can be an instance of KAP. This computation model[6] was done to allow a high degree of parallelisation.

4) Syntax of Oz

The Kernel of the language, as given in [8] is very small; this might be an advantage from the point of view of compilation/optimization. Furthermore, a small Kernel gives only a small number of keywords, and experience has shown that languages with too many keywords become harder to learn and read.

S	::=	`skip`	*empty statement*
	|	$S_1\ S_2$	*sequential and*
	|	`thread` S `end`	*concurrent compositions*
	|	`local` $X_1 \cdots X_n$ `in` S `end`	*variable introduction* $(n \geq 1)$
	|	$X{=}Y$	*imposing equality (binding)*
	|	$X{=}z$	
	|	$X{=}f(l_1{:}X_1 \ldots l_n{:}X_n)$	
	|	`if` X `then` S_1 `else` S_2 `end`	*conditional statements*
	|	`case` X `of` $f(l_1{:}X_1 \ldots l_n{:}X_n)$ `then` S_1 `else` S_2 `end`	
	|	`proc` $\{P\ X_1 \cdots X_n\}\ S$ `end`	*procedural abstraction*
	|	$\{P\ X_1 \cdots X_n\}$	
	|	$\{$`NewCell` $X\ C\}$	*state*
	|	$\{$`Exchange` $C\ X\ Y\}$	
	|	`try` S_1 `catch` X `then` S_2 `end`	*exception handling*
	|	`raise` X `end`	

A more complete syntax of Oz in BNF-style, taken from the lessons of Peter Van Roy at Université Catholique de Louvain, is given in the annexe at the end of this article.

[5] Parlog is a language derived from Prolog, where all the clauses are evaluated in parallel.

[6] For a deep understanding of the model of computation, see the Andorra Model, explained briefly in [7]. It has been proposed by D. H. D. Warren as a way to combine or-parellelism with dependent and-parellism, and has been implemented in the execution of Prolog programs. A full parallel execution of Prolog is done in one phase of the language Pandora, based on a generalisation of the Andorra model. Another variant of the Andorra Model (Extended Andorra Model) is used by the language KAP.

5) Computational Model of Oz

The Oz programming model [4,5,8] is an extension to a basic concurrent constraint model[7], adding first-class procedures and stateful data structures. A computation takes place in a computational space, hosting tasks connected to a shared store. The computation advances by atomic reduction of tasks; the process of reducing a task can manipulate the store and create new tasks. When a task is reduced, it disappears.

The store holds the information about program variables, such as "V is bound to 3" and "V is equal to W"[8]. The store is monotonic: information can only be added and no changed; as we will see, this has a strong implication in the declarative model.

The tasks impose constraints on variables; thus, the store can be considered as a logic conjunction of constraints. The store is also used to synchronize tasks[9] : they become reducible only if the store satisfies certain constraints[10].

The store provides two primitive operations :
· Tell. It adds information to the store so that the new store stays consistent. For example, telling W = 7 is not possible if the store has V=3^V=W.
· Ask. Synchronization mechanism on the information asked, which is called the *guard*.

The 'ask' instruction waits on the availability of the data asked. Thus, we have dataflow concurrency.

Related works on concurrent logic programming includes the language Strand[11], PCN and Id.

[7] Concurrent Constraint Programming (CCP) uses concurrency as the most general form of control and constraints as the way to communicate and synchronize. This model combines ideas from concurrent (*c.f.* E. Shapiro, *The Family of Concurrent Logic Programming* Languages, 1989) and constraint logic programming (*c.f.* Jaffar&Maher, *Constraint Logic Programming: A Survey*).

[8] These information are written in logic form.

[9] So far, we saw that we have a monotonic store, and tasks (or agents) communicating and synchronizing through this store. This is the basic definition of Concurrent Constraint Programming, thus Oz respects it fully.

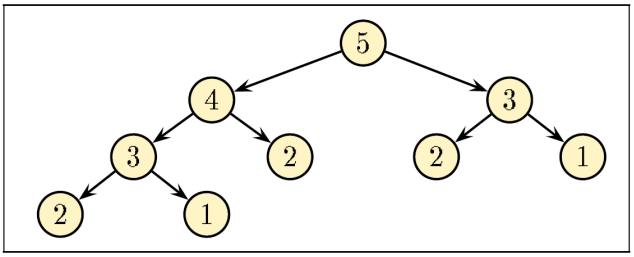
[10] According to Maher's, synchronization is a logical entailment between constraints.

[11] Strand (STReam AND-parallelism) was done by Ian Foster and Stephen Taylor, *c.f.* their book *Strand: New Concepts in Parallel Programming* (Prentice Hall, 1990).

6) *Efficient Reduction Strategy of Tasks*

Having an efficient reduction strategy is not trivial. The example given by Smolka is the recursion tree of a Fibonacci call : breadth-first strategy is exponential, whereas depth-first is in linear space. The reduction strategy organizes tasks into threads (fairness between tasks is therefore guarantee at the level of threads). Smolka describes his strategy as "as sequential as possible and as concurrent as possible"; it will execute the Fibonacci call in linear time complexity, instead of exponential.

```
proc {Fib N M}
   local B X Y in
      {Less 2 N B}
      if B=True then {Fib N-1 X}  {Fib N-2 Y}  X+Y=M else M=1 fi
   end
end
```



Calls in order to compute Fibonnaci of 5

If the user creates a function for Fibonacci with a new thread for each steps, then he breaks the strategy of Smolka and comes back to an exponential time :

```
fun {Fib X}
   case X
   of 0 then 1
   [] 1 then 1
   else thread {Fib X-1} end  + {Fib X-2} end
end
```

7) *Declarative Concurrency*

Oz is a concurrent and declarative language. The declarative aspect comes from the limitations of the monotonic store : once a variable has been declared, it is bound to a value and it cannot change; until a value is given, we consider a variable as being unbound. The basic of declarative programming is : tells the computer what you want to compute, and let it do it for you. A strict declarative programming is stateless : it does not allow an update of the data structures; however, we know that this might have some limitations, thus there are some way to go over it. When somebody does `declare Y = 13`, there is creation of a variable containing 13 in memory, and Y is bound to this memory.

8) *No modification but several declarations*

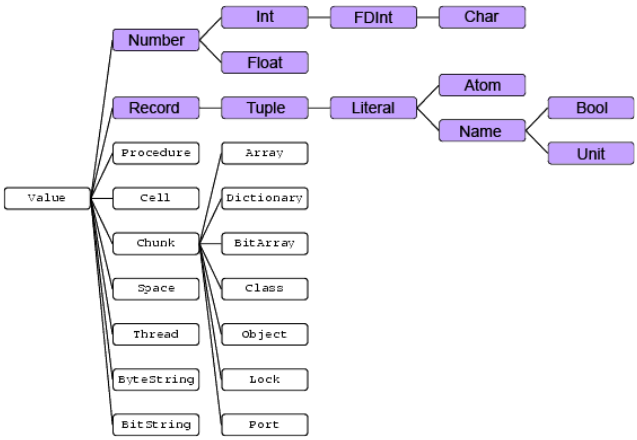
Once the user has done `declare Y = 13`, it is not possible to assign any other value to X. Thus, we will have an exception for `Y = 56`; for convenience, `Y = 13` will not do anything. The first solution if we want to modify the value of a variable is to declare another one of the same name : the environment will just refer to the last declaration, thus we will mask the older variable.

```
declare
Y = 13
declare
Y = 39
```

It is important to notice that this little trick does not break the paradigm : we are still declarative, and nothing forbid us to declare something with the same name more than once. Another way to allow modification will be through a data type named cells, which we will see in 3§2.

9) *Types*

Oz is a dynamically typed language : when a variable is introduced, its type and value are unknown; when the variable is bound to a value then its type is determined. The primary types available in Oz are shown in the scheme below.



Oz Type Hierarchy

The hierarchy starting from Number and Record (painted in purple on the scheme) uses structural equality[12]. Object, cells, chunks, etc. use name equality instead[13] and will be garbage collected.

[12] Values are equivalent if they have the same structure : they are made of the same basic elements, which have the same value. Thus, there is no way to distinguish between copies (useful for distributed computations): replicas occupying different memory location will have the same components in type and values, so they will be equal.

[13] Two variables are equal if they have the same identifier. Also found as 'token equality'. Values can be finalized.

3. Advanced notion of Oz

1) *Procedures*

In Oz, everything is a first-class entity, procedures included [5]. It means that a procedure can create a new procedure, have lexically scoped variables and can be referred by first-class values. A procedure has a name, any number of formal arguments and a body (*i.e.* expression). In the computational model of Oz, we saw a store for constraints; procedures are stored in another place, called the *procedure store*; for each name, this store contains at most one procedure and no procedure can be deleted. The values of the global variables[14] of the procedure are kept in the constraint store. The definition and application is very close to Lisp :

E	::=	**proc** $\{x\ z\}\ E$	*definition*
	|	$\{x\ y\}$	*application*

2) *Keeping state : introduction of Cells*

Beside the constraint store and the procedure store, there is a last store called *cell store*[15]. A cell is a *mutable* binding of a name to a variable: the reader will notice that for the first time, we have a possibility of changing directly something. Indeed, we can create a new cell, access its content and especially modifies it, using :

{NewCell X ?C}	*Creates a cell C with content X*
X=@C	*Stores the content of C in X*
C:=Y	*Modifies the content of C to Y.*
{IsCell +C}	*Test if C is a cell.*
{Exchange +C X Y}	*Swap atomically*[16] *the content of C from X to Y.*

Cells express stateful and concurrent data structures, which can be used as a way to communicate between concurrent agents. Built on the idea of cells, we have chunks and thus some abstract data types such as array and dictionary, or port (a communication channel). As we will see, a class is (in theory) stateless, and the state is encapsulated in the object.

[14] To understand the interface between variables and procedures, see *Fresh: A higher-order language with unification and multiple results* by G. Smolka, *in Logic Programming: Relations, Functions and Equations*, p469, Prentice Hall, 1986.

[15] In fact, there is only one big store holding three compartments: constraint, procedure and cells. However, we consider easier for the reader to have the idea of multiple stores, still within the same computation space.

[16] It is important to provide atomic operations in a concurrent environment, so that we can guarantee them.

3) *Basic of Objects*

A class has a collection of methods stored in a method table, a description of the attribute that each instance will posses; they will be stored in the instance as a stateful cell. A class being only a description of its objects, then classes are stateless in Oz, contrary to languages like Java. We use the following example from Oz tutorial :

```
declare Counter
local
   Attrs = [val]
   MethodTable = m(browse:MyBrowse init:Init inc:Inc)
   proc {Init M S Self}
      init(Value) = M in
      (S.val) := Value
   end
   proc {Inc M S Self}
      X inc(Value)=M
   in
      X = @(S.val) (S.val) := X+Value
   end
   proc {MyBrowse M=browse S Self}
      {Browse @(S.val)}
   end
in
   Counter = {NewChunk c(methods:MethodTable attrs:Attrs)}
end
```

Single attribute

methods

Environment of the method

The object can easily be created and manipulated:

```
declare C
{NewObject Counter init(0) C}
{C inc(6)} {C inc(6)}
{C browse}
```

This is the way that we could fake objects in Scheme as well, using closures; methods are procedures taking a message, the object itself (using Self), etc. However, Oz has a complete construction for classes with the keyword class :

```
class Counter
   attr val
   meth browse
      {Browse @val}
   end
   meth inc(Value)
      val := @val + Value
   end
   meth init(Value)
      val := Value
   end
end
```

The object will be created using the keyword New, and manipulated as before :

```
declare C in
C = {New Counter init(0)}
{C browse}
{C inc(1)}
```

4) Multiple Inheritance

Oz allows restricted multiple inheritance: using the keyword from, the programmer can inherit from as many classes as he wants as long as they do not share some methods names. If a method is defined at more than one superclass, the user has to define it locally. However, it is still possible to do multiple inheritance without respecting this constraint and be able to compile: classes are partially formed at compile time and are completed at execution time; thus, if the hierarchy is not correct, it will create a runtime exception.

5) Private and protected methods

Methods labelled by variables instead of literals are private, as shown in the Oz tutorial:

```
class C from ...
   meth A(X)          ...                end
   meth a(...)        {self A(5)} ...    end
   ....
end
```

There is direct to define a method as being protected; instead, it uses a trick: a method is made protected by making it private and storing it into an attribute. Thus, we get the following :

```
class C from ...
   attr pa:A
   meth A(X)          ...                end
   meth a(...)        {self A(5)} ...    end
   ...
end
```

6) Logic Programming in Oz

In the case of deterministic logic programming, we have the statements if, case and cond. Van Roy in [9] shows the similitude between the predicate append in Oz and its logical semantic:

```
declare
proc {Append L1 L2 L3}
   case L1
   of nil then L2=L3
   [] X|M1 then L3=X|{Append M1 L2}
   end
end
```

$$\forall l_1, l_2, l_3 : \text{append}(l_1, l_2, l_3) \leftrightarrow$$
$$l_1 = \text{nil} \wedge l_2 = l_3 \vee$$
$$\exists x, m_1, m_3 : l_1 = x|m_1 \wedge l_3 = x|m_3 \wedge \text{append}(m_1, l_2, m_3)$$

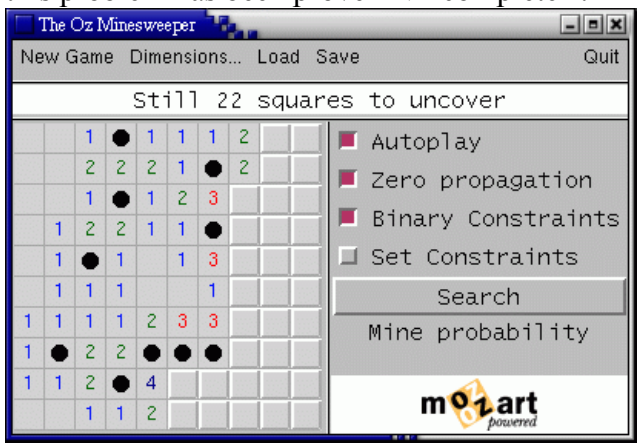
The statements dis and choice perform a search, and thus they enable nondeterministic logic programming.

4. Compilation references

AMOZ was the Abstract Machine underlying the implementation of Oz at DFKI in 1995, and interested readers can report to the related article [10], where most algorithms are explained, as well as the notions of threads and computation spaces. An introductory of the more modern Mozart Virtual Machine (MVM) is given in [11]. In a shorthand, MVM is a "*modern register-based virtual machine, with high-level bytecode instructions and built-in garbage collection routines [...] built in such a way that the compilers designers does not have to worry about many special details of the language, such as the Constraint Solving system and the lazy evaluation mechanism.*". The authors of [11] explain how they design the new compiler, from the front-end to the register allocation and optimizations steps.

5. Applications and developments

The Oz Minesweeper by Raphaël Collet uses constraint programming to implement a solver for the game; this is furthermore interesting as this problem has been proven NP-complete[17].



The P2PS library offers primitives and services for the development of peers to peers applications in Oz.

Oz is still in development. The Oz-E Project (part of EVERGROWN) wants to make Oz a secure language based on the principle of least authority.

Alice is presented as the successor with strong type-checking of Oz; constraint and concurrency are available. It is used by linguists.

[17] According to the claims of Richard Kaye.

Bibliography

[1] Gert Smolka, *The Development of Oz and Mozart*, in MOZ2004 (Charleroi, Octobre 2004)

[2] http://www.mozart-oz.org, page "governance"
[3] http://www.mozart-oz.org, page "Tutorial of Oz" (Seif Haridi & Nils Franzen), in "documentation"

[4] Martin Müller, Tobias Müller, Peter Van Roy, *Multiparadigm Programming in Oz*, 1995

[5] Gert Smolka, *The Oz Programming Model (OPM)*, July 11, 1995

[6] S. Janson, S. Haridi, *Programming Paradigms in the Andorra Kernel Language (AKL)*, 1991

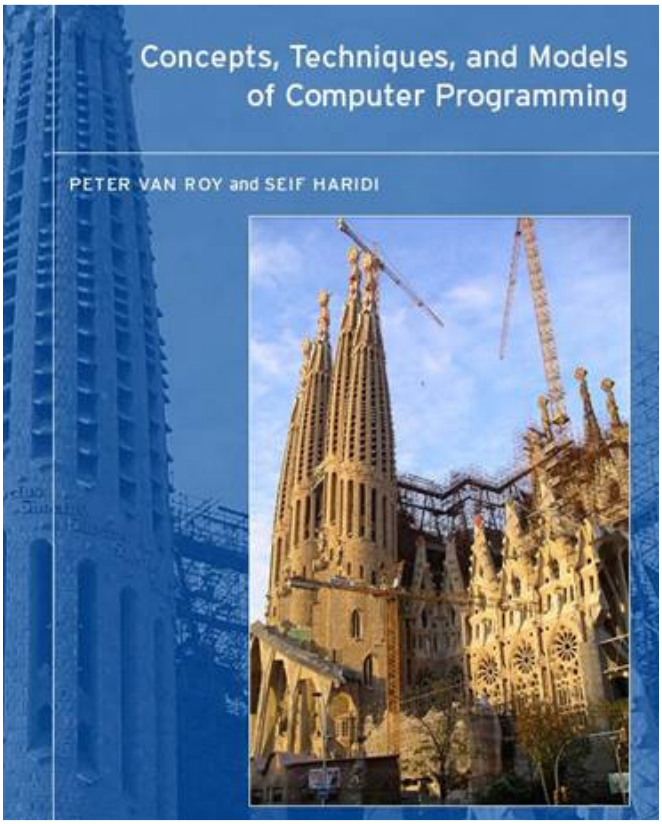
[7] Sverker Janson, Seif Haridi, *Kernel Andorra Prolog and its Computation Model*, November 13, 1992

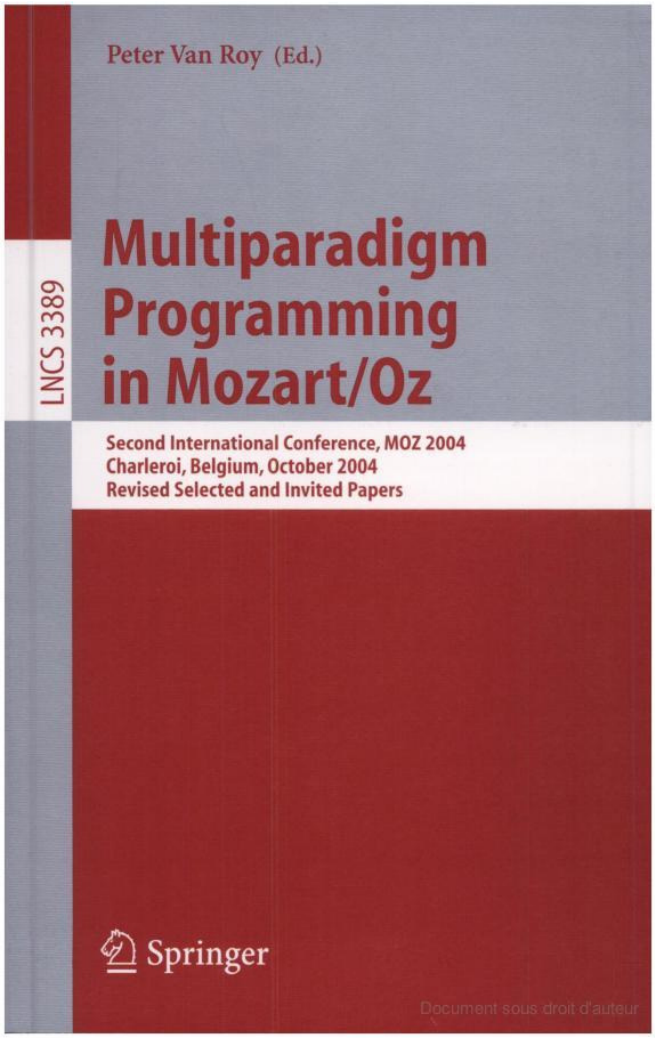
[8] Raphaël Collet, Peter Van Roy, *The Operational Semantics of Oz*, March 7, 2001

[9] Peter Van Roy, *Logic Programming in Oz with Mozart*, 1999

[10] M. Mehl, R. Scheidhauer, C. Schulte, *An Abstract Machine For Oz*, DFKI, June 27, 1995

[11] Markus Bohlin, Lars Bruce, *Redesign of the Oz Compiler*, Master Thesis, June 12, 2002





Annexe : Syntax of Oz

```
<statement> ::= skip
              | <statement> <statement>
              | <variable> = <expression>
              | ( <in statement> )  |  local {<declaration>}+ in <statement> end
              | if <expression> then <in statement>
                { elseif <expression> then <in statement> }
                [ else <in statement> ]
                end
              | case <expression>
                of <pattern> [ andthen <expression> ] then <in statement>
                { [] <pattern> [ andthen <expression> ] then <in statement> }
                [ else <in statement> ]
                end
              | {<expression>  {<expression>} }
              | proc {<variable>  {<pattern>} } <in statement> end
              | fun {<variable>  {<pattern>} } <in expression> end
              | <exception statement>  |  <stateful statement>  |  <functor statement>
<in statement> ::= [ {<declaration>}+ in ] <statement>
<declaration> ::= <variable>  |  <pattern> = <expression>  |  <statement>
```

```
<exception statement> ::= try <in statement>
                          [ catch <pattern> then <in statement>
                            { [] <pattern> then <in statement> } ]
                          [ finally <in statement> ]
                          end
                        | raise <in expression> end
<exception expression> ::= try <in expression>
                           [ catch <pattern> then <in expression>
                             { [] <pattern> then <in expression> } ]
                           [ finally <in statement> ]
                           end
                         | raise <in expression> end
```

```
<expression> ::= <variable>  |  _
               | <data expression>
               | <unary operator> <expression>
               | <expression> <binary operator> <expression>
               | ( <in expression> )
               | local {<declaration>}+ in [<statement>] <expression> end
               | if <expression> then <in expression>
                 { elseif <expression> then <in expression> }
                 [ else <in expression> ]
                 end
               | case <expression>
                 of <pattern> [ andthen <expression> ] then <in expression>
                 { [] <pattern> [ andthen <expression> ] then <in expression> }
                 [ else <in expression> ]
                 end
               | { <expression>  {<expression>} }
               | proc {$  {<pattern>} } <in statement> end
               | fun {$  {<pattern>} } <in expression> end
               | <exception expression>  |  <stateful expression>  |  <functor expression>
<in expression> ::= [ {<declaration>}+ in ] [<statement>] <expression>
<unary operator> ::= ~
<binary operator> ::= . | + | - | * | / | div | mod | == | \= | < | =< | > | >= | andthen | orelse
```

```
<functor statement> ::= functor <variable>
                        [ import { <variable>  [at <atom>] }+ ]
                        [ export { [<feature>:] <variable> }+ ]
                        define {<declaration>}+ [in <statement>]
                        end
<functor expression> ::= functor [$]
                         [ import { <variable>  [at <atom>] }+ ]
                         [ export { [<feature>:] <variable> }+ ]
                         define {<declaration>}+ [in <statement>]
                         end
```

```
<data expression> ::= <atom> | <integer> | <float> | <string> | <oz character> | true | false | unit
                    | <label> ( { [<feature> :] <expression> } )
                    | <expression>|<expression>  |  <expression> {#<expression>}+  |  [ {<expression>}+ ]
<pattern> ::= <variable>  |  _
            | <atom> | <integer> | <float> | <string> | <oz character> | true | false | unit
            | <label> ( { [<feature> :] <pattern> }  [...] )
            | <pattern>|<pattern>  |  <pattern> {#<pattern>}+  |  [ {<pattern>}+ ]
            | ( <pattern> )
<label> ::= <atom> | true | false | unit
<feature> ::= <atom> | <integer> | true | false | unit
```

```
<variable> ::= <upper-case letter> {<alphanumerical>}
             | ` {<character>} `
<atom> ::= <lower-case letter> {<alphanumerical>}
         | ' {<character>} '
<integer> ::= [~] ( 0 | <non-zero digit> {<digit>} )
<float> ::= [~] {<digit>}+. {<digit>} [ (e | E) [~] {<digit>}+ ]
<string> ::= " {<character>} "
<oz character> ::= & <character>
```

```
<stateful statement> ::= <expression> := <expression>  |  <expression>,<expression>
                       | class <variable>
                         { <class descriptor> }
                         { <method> }
                         end
<stateful expression> ::= {NewCell <expression>}  |  @<expression>  |  self
                        | class [$]
                          { <class descriptor> }
                          { <method> }
                          end
<class descriptor> ::= from {<expression>}+  |  attr {<feature>}+
<method> ::= meth <method head> <in statement> end
<method head> ::= <label> ( { [<feature> :] <variable> [<= <expression>] }  [...] )
```

The reader could also be interested in the attempt of Tobias Nurmiranta to use a Scheme syntax for Oz semantics, thanks to a little file that can be loaded into Scheme programs.

The syntax given on this annexe is not complete; it describes only a part of the language used in the lessons of Peter Van Roy at Université Catholique de Louvain (Belgium).