# Declarative programming with Oz

November 1, 2017

The purpose of this lab is to familiarize you with the Mozart programming system, and how to program in Oz.

### What is Oz?

Oz is a multiparadigm programming language, developed for programming language education. It integrates programming features for

- logic programming
- functional programming: both lazy evaluation and eager evaluation
- imperative, object-oriented, constraint, distributed, and concurrent programming.

The main textbook which describes Oz is

- P. Van Roy, S. Haridi: Concepts, Techniques, and Models of Computer Programming. MIT Press. 2004.

The Mozart Programming System is an open source implementation of Oz 3. It was developed by researchers from DFKI (the German Research Center for Artificial Intelligence), SICS (the Swedish Institute of Computer Science), the University of the Saarland, UCL (the Université catholique de Louvain), and others. Mozart is freely available from the github repository `http://mozart.github.io`

## Downloads

Binary packages for Linux, Mac OS and Windows are built from time to time and made available on SourceForge[1]:

`https://sourceforge.net/projects/mozart-oz/files/`

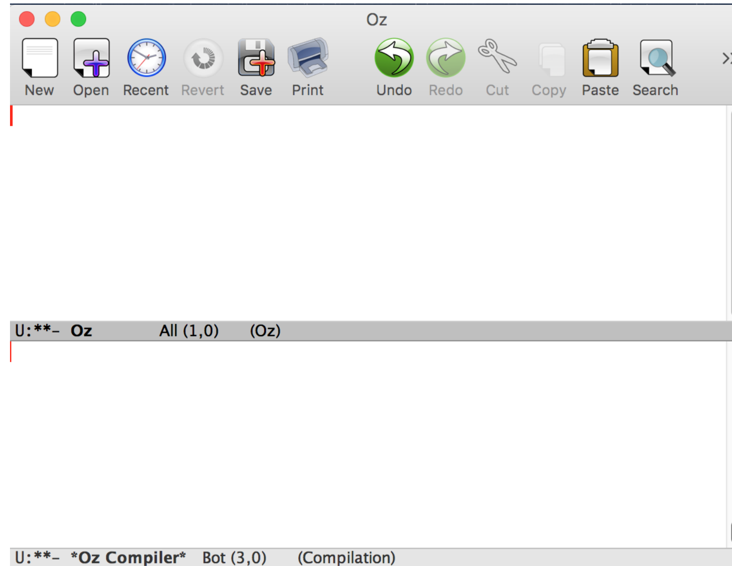
Mac support is provided for 10.8.x and recent versions (2.x) of Aquamacs.

The binary distribution requires that you have installed Tcl/Tk 8.5 on your system.

[1] sourceforge.net is the largest Open Source applications and software directory

# 1 Getting started

Double-click the icon of `Mozart` on your desktop. This opens an editor window with two frames (or text buffers):[2]




The upper buffer, called `Oz`, is where you can write small pieces of code and execute them interactively. From now on, we can use Oz to do calculations:

- Use the top frame to type statements in the syntax of Oz.
- Evaluate the instructions you typed in. There are two ways to do so:
  1. Go to the `Oz` menu and select
     - `Feed Region`, to feed the selected text to the system.
     - `Feed Line`, to feed the text from the current line to the system.
     - `Feed Paragraph`, to feed the last paragraph of text to the system.
  2. The alternative is to press
     `Ctrl`+`.`+`r` instead of `Feed Region`
     `Ctrl`+`+.`+`l` instead of `Feed Line`
     `Ctrl`+`Alt`+`x` instead of `Feed Paragraph`

The syntax of Oz instructions was explained in the lecture notes. See Appendix A to remember it.

[2]Mozart uses the Emacs editor as the programming frontend.

## 1.1 Doing some simple computations

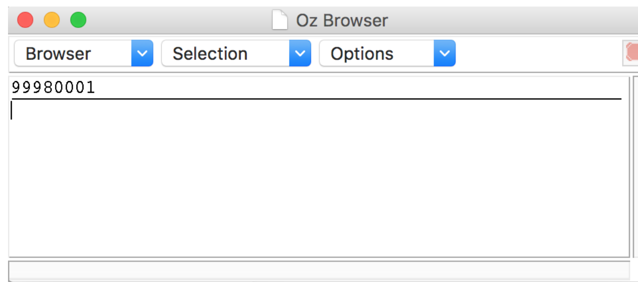
Let's compute `9999*9999`, and view its value:

> `9999*9999` is a function call: it is syntactic sugar for
>
> `{Number.'*' 9999 9999}`

By default, the system does not prints the results of evaluating expressions. To print them, use the function `Browse`:

```
{Browse 9999*9999}
```

After doing the calculations, Oz displays the result in a special window called the browser:



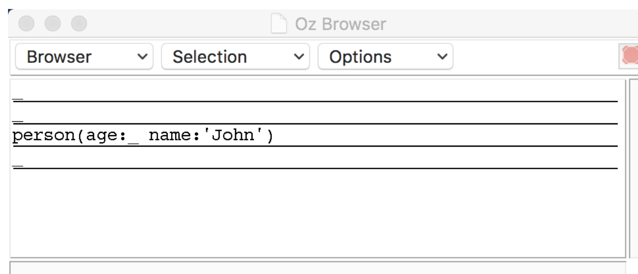

Note that `{Browse X}` opens the browser window, if it is not already open, and displays the value of `X` in it.

## 1.2 Variables and variable-vriable bindings

Variables are placeholders for values. Every variable must be declared before being used. Variables can be bound to values when they are declared, or they can get bound afterwards. For example, you can type and evaluate the following statements in the `Oz` frame:

```
declare X Y Z=person(name:'John' age:X) T
{Browse X} {Browse Y} {Browse Z} {Browse T}
```

The first statement declares 4 variables $x, y, z, t$, makes `X` refer to $x$, `Y` to $y$, `Z` to $z$, `T` to $t$, and binds $z$ to the partial value `person(name:'John' age:`$x$`)`. The following three calls of `Browse` make the browser look as follows:

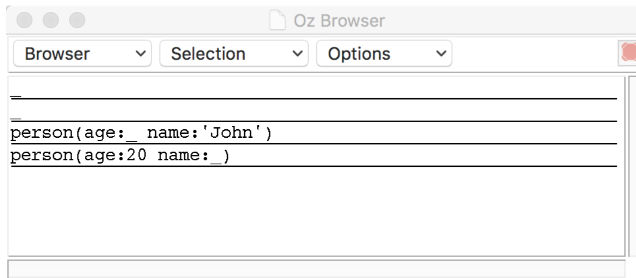


The browser indicates that `X`, `Y`, `T` refer to unbound variables, and `Z` refers to the partial record value `person(age:`$x$ `name:'John')`

If we feed the statement

```
T = person(name:Y age:20)
```

then Oz binds $t$ to value `person(age:20 name:`$y$`)`. The browser gets notified about the change in the store and updates the displayed values:



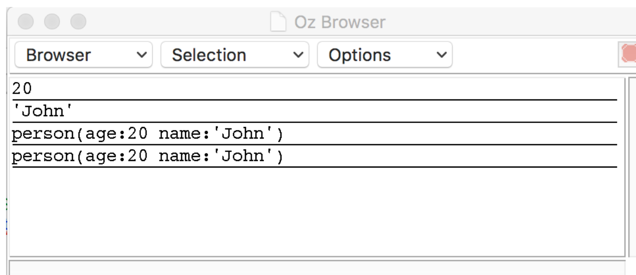

If we feed

```
Z = T
```

then Oz tries to make compatible the values of `Z` and `T`, which are

    person(age:𝑥 name:'John')
    person(age:20 name:𝑦)

As a result, $x$ gets bound to `20`, and $y$ to `'John'`. The browser will display




Some values are incompatible: If we feed `Z = person(age:21 name:'John')` then Oz detects that `person(age:21 name:'John')` is incompatible with the value of `Z`, and reports the failure of an equality constraint in the second frame:

```
%********************** static analysis error *******************
%**
%** equality constraint failed
%**
%** First value:        20
%** Second value:       21
%** Original assertion: person(age:20 name:'John') = person(age:21 name:'John')
%** in file "Oz", line 7, column 3
%** ------------------ rejected (1 error)
```

## 1.3 Functions

We can define functions and use them later. For example, the factorial function can be defined recursively as follows:

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

Factorial values are very large numbers, but Oz knows how to cope with them. If we feed

```
{Browse {Fact 100}}
```

the browser will display a number with 358 decimal digits. This is an example of arbitrary precision arithmetic: The precision is limited by how much memory your system has.

Now, we can reuse `Fact` to define other functions, for example

```
declare
fun {Comb N K}  /* computes the value of C(n,k) */
   {Fact N} div ({Fact K}*{Fact N-K})
end
```

## 1.4 Working with lists

We illustrate this capability by looking at Pascal's triangle. Pascal's triangle is a key concept in combinatorics: it consists of an infinite numbers of rows, starting from row 0. For every $n \in \mathbb{N}$, the the $n$-th row of the triangle is a list whose elements are the combinations $\mathrm{C}(n,k)$, where $k$ ranges from 0 to $n$. For example, row 0 is the list `[1]` and row 1 is the list `[1 1]`.

A nice property of Pascal's triangle is the following: For every $n > 0$, the $n$-th row can be computed recursively from the $(n-1)$-th row as follows:

- Shift the $(n-1)$-th row to the right by adding element 0 in front of it $\Rightarrow$ a list `R` with $n$ elements
- Shift the $(n-1)$-th row to the left by adding element 0 in front of it $\Rightarrow$ a list `L` with $n$ elements
- The $n$-th row is the componentwise addition of lists `L` and `R`

For example, he third row can be computed from the second row `[1 1]` as follows:

- the right shift of `[1 1]` is `R=[0 1 1]`
- the left shift of `[1 1]` is `L=[1 1 0]`
- The componentwise addition of lists `L` and `R` yields the second row `[1 2 1]`.

Let's define a function such that `{Pascal N}` returns the `N`-th row of Pascal's triangle:

```
declare fun {ShiftRight L}
   0|L
end

declare fun {ShiftLeft L}
   case L of nil then [0]
   [] H|T then H|{ShiftLeft T}
   end
end

declare fun {AddList L1 L2}
   case L1#L2 of nil#nil then nil
   [] (L|Ls)#(T|Ts) then (L+T)|{AddList Ls Ts}
   end
end

declare fun {Pascal N}
   if N==0 then [1]
   else {AddList {ShiftLeft {Pascal N-1}}
                 {ShiftRight {Pascal N-1}}}
   end
end
```

This definition is logically correct: it works, but the implementation is very inefficient

```
{Browse {Pascal 23}}
```

will eventually display the result

```
[1 23 253 1771 8855 33649 100947 245157 490314 817190
1144066 1352078 1352078 1144066 817190 490314 245157
100947 33649 8855 1771 253 23 1]
```

but it will take long time (explain why!).

The following alternative definition is much faster:

```
declare fun {FastPascal N}
   if N==0 then [1]
   else L in
          L = {FastPascal N-1}
          {AddList {ShiftLeft L} {ShiftRight L}}
   end
end
```

## 1.5 Lazy evaluation

There are two ways to do calculations:

**strict:** functions do their calculation as soon as they are called

**lazy:** functions do their calculation only when the result is needed.

Most functional programming languages are either strict (e.g., Racket, Common Lisp) or lazy (e.g., Haskell). Oz can do both strict and lazy calculations.

The following lazy function computes the list of all numbers starting from a given integer:

```
declare
fun lazy {Ints N}
   N|{Ints N+1}
end
```

Note that the strict version of this function definition is useless (explain why!):

```
declare
fun {StrictInts N}
   N|{StrictInts N+1}
end
```

The function call `{Ints 0}` is expected to calculate the infinite list `0|1|2|3|`... The `lazy` annotation ensures that the function call will be evaluated only when it is needed:

```
declare Nats = {Ints 0}
{Browse Nats}
```

The value for `Nats` displayed in the browser is `_`. This is so because the browser does not cause lazy functions to be evaluated.

If some elements of `Nats` are needed, then this function will be called automatically. For example, if `L` is a list, then

- The expression `L.1` needs to return the first element of list `L`,
- The expression `L.2` needs to return the tail of list `L`.

Therefore, if we feed

```
Nats.2.2.1
```

we inform the system that we need the first element of the tail of tail of `L`, that is, the third element of `Nats` (which is `2`). As a result, the value of `Nats` will be calculated until it has the first 3 elements, because this is the minimum calculation from which we can retrieve the 3rd element. After this calculation, the value of `Nats` becomes `0|1|2|{Nats 3}` and the browser displays `0|1|2|_` We can define more intricate lazy functions. For example, we can define a function to compute lazily the Pascal's triangle, that is, the infinite list $row_0|row_2|row_2|\ldots$ where $row_n$ is the $n$-th row of Pascal's triangle:

```
declare
fun lazy {PascalList Row}
    Row|{PascalList {ShiftLeft Row} {ShiftRight Row}}
end

declare PascalTriangle = {PascalList [1]}
```

If we need the fifth row of Pascal's triangle, which is, we can feed

```
PascalTriangle.2.2.2.2.1
```

After this, `PascalTriangle` refers to the value

```
[1]|[1 1]|[1 2 1]|[1 3 3 1]|[1 4 6 4 1]|{PascalList [1 4 6 4 1]}
```

## 1.6 Concurrency

Concurrency is the capability to perform several independent activities simultaneously, without interference, unless the programmer decides that the activities need to communicate.

Oz supports concurrency by creating threads. A program can have many threads that can run simultaneously. Threads are created with the `thread` instruction.[3] For example

```
thread P in
    P = {Pascal 30}   /* time-consuming calculation */
    {Browse P}
end
{Browse 99*99}        /* very short calculation */
```

This program creates a new thread which performs a time-consuming calculation. The new thread will not stop the main thread of the program to run and display the value of `99*99` immediately.

## References


Chapter 1: *Introduction to Programming Concepts* from the book

P. Van Roy, S. Haridi: Concepts, Techniques, and Models of Computer Programming. MIT Press. 2004.


[3]`thread` is not part of the language of declarative programming. It is part of the larger language of Oz, which integrates many programming paradigms.

# A The syntax of Oz statements

Oz consists of kernel language + syntactic sugar + linguistic abstractions.

The kernel language is very small: it has just 8 kinds of statements:

1. `skip`
   The empty statement. It does nothing.
2. $\langle s\rangle_1$ $\langle s\rangle_2$

   Sequence of statements $\langle s\rangle_1$ and $\langle s\rangle_2$, which are executed one after the other. Note that statements are separated by whitespace.
3. `local` $\langle x\rangle$ `in` $\langle s\rangle$ `end`

   Creates the new unbound variable $x$, lets identifier $\langle x\rangle$ refer to variable $x$, and then evaluates statement $\langle s\rangle$
4. $\langle x\rangle_1 = \langle x\rangle_2$

   Variable-variable binding. Constrains the values of variables $\langle x\rangle_1$ and $\langle x\rangle_2$ to be compatible in the store.
5. $\langle x\rangle = \langle v\rangle$

   Variable binding. It creates a new variable $x$ in the store, binds $x$ to the value of $\langle v\rangle$ in the store, and lets $\langle x\rangle$ refer to $x$.
6. `if` $\langle x\rangle$ `then` $\langle s\rangle_1$ `else` $\langle s\rangle_2$ `end`

   Conditional.
7. `case` $\langle x\rangle$ `of` $\langle pattern\rangle$ `then` $\langle s\rangle_1$ `else` $\langle s\rangle_2$ `end`

   Pattern matching.
8. $\{\langle x\rangle\ \langle y\rangle_1\ \ldots\ \langle y\rangle_n\}$

   Procedure call.

In Oz, there are at least 3 kinds of values:

$\langle v\rangle$ `::=` $\langle number\rangle$ | $\langle record\rangle$ | $\langle procedure\rangle$ | $\ldots$
$\langle number\rangle$ `::=` $\langle int\rangle$ | $\langle float\rangle$
$\langle record\rangle, \langle pattern\rangle$ `::=` $\langle literal\rangle$
| $\langle literal\rangle(\langle feature\rangle_1 : \langle x\rangle_1\ \ldots\ \langle feature\rangle_n : \langle x\rangle_n)$
$\langle procedure\rangle$ `::= proc` $\{\$\ \langle x\rangle_1\ \ldots\ \langle x\rangle_n\}$ $\langle s\rangle$ `end`
$\langle literal\rangle$ `::=` $\langle atom\rangle$ | $\langle bool\rangle$
$\langle feature\rangle$ `::=` $\langle atom\rangle$ | $\langle bool\rangle$ | $\langle int\rangle$
$\langle bool\rangle$ `::= true` | `false`

where $\langle x\rangle, \langle y\rangle, \langle x\rangle_1, \ldots, \langle x\rangle_n, \langle y\rangle_1, \ldots, \langle y\rangle_n$ are variable identifiers. A variable identifier must either

- start with an uppercase letter, like in Prolog. For example, `X`, `Y123`, `Zflag` are variable identifiers; or
- be a sequence of characters between backquote (`) characters. For example, `` `|` `` and `` `this is a variable id` `` are variable identifiers.