Oz demo

September 9th

Mark Armstrong, PhD Candidate, McMaster University

Fall, 2019

Contents

1 Getting started

The Mozart programming system uses Emacs as an editor, with dedicated keybindings for interacting with the Oz virtual machine.

In an Oz buffer (window), use the key sequence `C-. C-b` (that is, `Ctrl-.` followed by `Ctrl-b`) to "feed" the buffer to the virtual machine.

Another useful command is `C-. h`, to *halt* the virtual machine in case of any issue. Note this will lock Emacs up for 30 seconds!

The "Tutorial of Oz", even though it is for the older Mozart v1, is of great use when learning Oz. It was my primary resource when writing these notes.

1.1 Using Emacs help: most things are documented!

For more details on using Oz in Emacs, type `C-h o` and then `oz-mode` followed by `enter`.

Alternatively, type `M-x describe-mode` (where `M-x` is "Meta-X"; usually Meta is `Alt`, so `M-x` is `Alt-x`) followed by `oz-mode` and then `enter`.

This will open the `*Help*` buffer (window) describing `oz-mode`, which has a list of all relevant keybindings and links to more help pages (documentation).

In Emacs, almost every mode, function and value is documented in this way. To see the different kinds of help you can get, type `M-x describe` and then `tab`, which whill show you all of the `describe` functions.

Of particular help are `describe-symbol` (`C-h o`) and `describe-key` (`C-h k`).

1.2 Want to set up Oz in your Emacs environment?

If you use Emacs outside of just using Oz, you may be annoyed that Oz is not set up when you launch Emacs without using the Oz launcher.

Adding the below elisp code to your Emacs init file should solve that problem. This is taken from my Emacs init, where I explain what's going on. (It's not been tested on other machines, so YMMV).

```
(setq my-oz-home "/usr")

(when (file-directory-p my-oz-home)
  (setenv "OZHOME" my-oz-home)
)
```

```
(setq my-mozart-elisp "/usr/share/mozart/elisp")

(when (file-directory-p my-mozart-elisp)
  (add-to-list 'load-path my-mozart-elisp)
  (load "mozart")
  (add-to-list 'auto-mode-alist '("\\.oz\\'" . oz-mode))
  (add-to-list 'auto-mode-alist '("\\.ozg\\'" . oz-gump-mode))
  (autoload 'run-oz "oz" "" t)
  (autoload 'oz-mode "oz" "" t)
  (autoload 'oz-gump-mode "oz" "" t)
  (autoload 'oz-new-buffer "oz" "" t)
)

(eval-after-load "oz-mode"
  '(progn
    (define-key oz-mode-map (kbd "C-x SPC") 'rectangle-mark-mode)
))
```

1.3 Want to write literate Oz code in Org mode?

If you are using Org mode, you may wish to write literate Oz code in Org.

Thankfully, support for evaluating Oz code blocks is built-in in Org. Simply add this to your init file.

```
(require 'ob-oz)
```

Then you write Oz code blocks like

```
#+begin_src oz :results output :tangle yes
{Browse 'Hello world'}
#+end_src
```

`:results` must be either `output` or `result`. You probably also want the option `:tangle yes` to produce raw source files.

To split your code up amongst several code blocks, you can use `:noweb yes`. This lets you import other code blocks by their name. For example,

```
#+name: define-x
#+begin_src oz :results output :noweb yes
declare
  X = 2
#+end_src
```

```
#+begin_src oz :results output :noweb yes
<<define-x>>

{Browse X}
#+end_src
```

2 Hello worlds

Let's begin with the simplest hello world. Just display a sequence of characters.

```
{Browse 'Hello world'}
```

We can make it a bit more interesting with a (immutable) variable (i.e. a constant). We precede variable declarations with the `declare` keyword.

```
declare
  S = 'Hello world'

{Browse S}
```

We can use arbitrary sequences of characters as names, surrounded by backticks.

```
declare
`A fancier greeting` = 'A pleasure to make you acquaintance, my good fellow.'

{Browse `A fancier greeting`}
```

3 Some datatypes

So far we've worked only with Oz's *atoms* (not strings!). An atom may be:

- A sequence of characters, numbers and underscores beginning with a lowercase letter.
  - Not a language keyword!
- A sequence of printable characters enclosed in single quotes.
  - The body of the atom cannot include single quotes, for obvious reasons.

```
declare
`An alphanumeric atom` = this_Is_An_Atom
`A printable characters atom` = 'This is also an atom.'
`A quoted keyword` = 'if'

{Browse `An alphanumeric atom`}
{Browse `A printable characters atom`}
{Browse `A quoted keyword`}
```

Oz of course includes numeric datatypes

```
{Browse 123}   % integer
{Browse 1.23}  % floating point
{Browse &{ }   % character ({ is 123 in ASCII)
{Browse 0123}  % octal (leading 0)
{Browse 0x123} % hexidecimal (leading 0x)
```

Oz has booleans; `true` and `false` are not atoms, they are special, reserved names. (Reserved names actually form a type, which we may investigate later).

```
{Browse true}
{Browse false}
```

Oz has lists, using a syntax similar to Prolog.

- `nil` is the empty list.
- `|` prepends a single element to a list; it is read "cons".
- brackets, `[]`, are included as *syntactic sugar*. No commas!

```
{Browse nil}
{Browse (1 | 2 | 3 | nil)}
{Browse [1 2 3]}
```

Strings are lists of characters.

```
{Browse "Hello world"}
{Browse [&H &e &l &l &o &  &w &o &r &l &d]}
{Browse ""}
```

Oz has *records* for packing data together.

```
declare
  X = my_record_name(
        descr : 'A record carrying this string and an integer'
        a_num : 123)

{Browse X}
{Browse X.descr}
{Browse X.a_num}
```

If you don't name fields, they're given numerical names.

```
declare
X = r('I am field 1. Field 2 is 123. Field 3 is true.' 123 true)

{Browse X.1}
{Browse X.2}
{Browse X.3}
```

Lists are implemented as records; `1` is the head and `2` is the tail.

```
declare
  L = [1 2 3]
  `Explicit L` = '|'(1 '|'(2 '|'(3 nil)))

{Browse L.1}
{Browse L.2}
{Browse `Explicit L`}
```

4 Control structures

For now, we'll consider branching structures only. Loops are not much use until we have mutable state!

Oz has `if` statements.

```
{Browse (if true then 'true is true.' end)}
```

There is an optional `else` clause. A runtime error will occur if it is needed and missing.

```
{Browse (if true then 'First branch' else 'Second branch' end)}
```

General rule: always include an `else` for expressions. It can be optional for statements.

```
if false then {Browse 'No else clause here.'} end
```

Oz also has the `elseif` syntactic sugar.

```
{Browse (if false then 'No.'
     elseif false then 'No.'
                  else 'Yes.'
        end)}
```

And Oz includes a case statement for pattern matching. This is especially useful for working with lists! (Note that `_` indicates an anonymous variable).

```
declare
  L = [1 2 3]

{Browse (case L of
               nil       then 'Empty'
            [] _|nil     then 'Singleton'
            [] _|_|nil   then 'Length 2'
            [] _|_|_|nil then 'Length 3'
                         else 'Longer than 3'
        end)
}
```

5 Comparisons

Oz includes equality and other comparison operators.

```
{Browse 1 == 1}
{Browse 1 =< 1}
{Browse 1 >= 1}
{Browse 1 <  1}
{Browse 1 >  1}
```

They can be applied to other types. Most types have equality.

```
{Browse atom1     == atom2    }
{Browse "String1" == "String2"}
{Browse [1 2 3]   == [1 2 3 4]}
{Browse true      == false    }
{Browse r(1 2 3)  == r(2 3 4) }
```

Atoms are lexicographically ordered.

```
{Browse abc < bcd}
{Browse abc < abcd}
```

Booleans, strings, lists and records are not ordered.

```
% {Browse [1] < [2]}   % Cannot compare lists!
```

6 Calling functions/procedures

Function/procedure calls are indicated with braces, as in `{F x y ...}`. *Braces are not for grouping*; parentheses may be used for that. (Though parentheses are often not necessary).

```
{Browse {And false false}}
{Browse {And false true }}
{Browse {And true  false}}
{Browse {And true  true }}
```

Note: we've been doing this all along with `Browse`, a procedure!

7 Expressions and statements

An *expression* has a value, and not (necessarily) not a side effect. A *statement* has a side effect, and not (necessarily) a value.

So far, the only statement we have used is `{Browse ...}`. Everything else has been an expression.

In Oz, an expression cannot be used where a statement is expected, and vice versa. This gives an error:

```
%1   % Cannot "execute" an expression!
```

A *function* is a subroutine which produces a value; therefore, a function invocation is an expression.

A *procedure* is a subroutine which produces a side effect; therefore, a procedure invocation is a statement.

For now, let's focus on functions.

8 Defining functions

Like with (immutable) variables, function declarations are preceded by a `declare` keyword. The syntax for a function declaration is `fun {FunctionName Arg1 Arg2 ...} FunctionBody end`. `FunctionBody` must end with an expression, and its value will be the "return value".

```
declare
  fun {Exp M N}
    if N == 0 then 1 else M * {Exp M (N - 1)} end
  end

{Browse {Exp 2 5}}
```

Question; what is this, really?

```
declare
  fun {F}
    5
  end
```

The `FunctionBody` may involve statements (and so it can have side effects). After the statements must come an expression.

```
declare
  fun {F}
    {Browse 10} % do something
    10          % return a value
  end
```

9 Aside: currying

The above `Exp` function is not curried; we cannot partially apply it. We can write a curried version using a *nested function definition*. We do not name the inner function (it is anonymous); we write `$` in place of a name.

```
declare
fun {Exp M}
  fun {$ N}
    if N == 0 then 1 else M * {{Exp M} (N - 1)} end
  end
end
```

```
{Browse {{Exp 5} 5}}
```

A curried function has to be partially applied!

10 Aside: large numbers

Note that we have "infinite" precision arithmetic in Oz. It's limited only by the physical limit of our machines.

```
declare
fun {Exp M N}
  if N == 0 then 1 else M * {Exp M (N - 1)} end
end

{Browse {Exp 5 5000}}
```

11 Aside: mutual recursion

What if we want to define two functions which each depend upon the other? That's allowed!

```
declare
  fun {Step2 M}
    if M > 1 then {Step1 M - 2} else M end
  end

  fun {Step1 M}
    if M > 0 then {Step2 M - 1} else M end
  end

  fun {MyPlan M}
    {Step1 M}
  end
```