Oz CheatSheet

Oz provides the *harmonious* support for many paradigms; e.g., OOP, FP, Logic, concurrent and networked. Moreover, every entity in Oz is first-class; e.g., classes, threads, and methods.

- Oz is a dynamically typed language, but strongly so: No coversions are performed; e.g., condition `5.0 = 5` raises an exception.
- It is strong in that

Setup

Download & install prebuilt binary.

```
# Ubuntu:
wget https://github.com/mozart/mozart2/releases/download/v2.0.1/mozart2-2.0.1-x86_64-linux.deb
sudo apt install ./mozart2-2.0.1-x86_64-linux.deb
oz

# Mac OS:
brew tap dskecse/tap
brew cask install mozart2
mozart2
```

Emacs setup —trying to accommodate Ubuntu and Mac OS.

```
;; C-h o system-type ⇒ See possible values.
;; darwin ⇒ Mac OS
(setq my-mozart-elisp
      (pcase system-type
       ('gnu/linux "/usr/share/mozart/elisp")
       ('darwin    "/Applications/Mozart2.app/Contents/Resources/share/mozart/elisp")))

;; Mac OS needs to know the location.
(add-to-list 'exec-path "/Applications/Mozart2.app/Contents/Resources/")

(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))

;; oz-mode annoyingly remaps C-x SPC, so we must undo that.
(eval-after-load "oz-mode"
  '(define-key oz-mode-map (kbd "C-x SPC") 'rectangle-mark-mode))

;; Org-mode setup for Oz; the Oz browser needs output.
(require 'ob-oz)
(setq org-babel-default-header-args:oz '((:results . "output")))
```

In an Emacs org-mode source block, executing the following brings up an Oz window —as desired.

```
declare X = 12
{Browse X}
```

All subsequent calls to `Browse` will output to the same window, unless it's closed.

Instead, we may use `Show` and have output rendered in the Emacs buffer `*Oz Emulator*`.

```
{Show 'Hello World'}
```

Jargon:

Oz The programming language at hand.

Mozart The implementation of Oz.

OPI The Oz Programming Interface, "OPI", which is built-around Emacs.

Variables

Names that begin with a capital letter; a `declare` close affects all following occurrences and so is 'global'.

```
declare V = 1
{Show V}          % ⇒ 1
```

```
declare V = 2
{Show V}          % ⇒ 2
```

One may also make local declarations; e.g., `local X Y Z in S end`.

Functions

Function application is written `{F` X_1 `...` X_n`}` —without parenthesis!

- This approach is inherited from Lisp.
- The last expression in the function body is its "return value", unless declared otherwise.
- If you write `{F(X)}` you will obtain a `illegal record label` error since `F` is a function name, not a literal.
- Use parenthesis only on compound expressions, which is seldom needed since *infix operators bind strongest*.

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

declare fun {Mult X Y} X * Y end
{Show {Fact Mult 5}}  % ⇒ 120

% Using an anonymous function.
{Show {Fact fun {$ X Y} X + Y end 5}}  % ⇒ 6

% Two ways to invoke a function.
{Show {Mult 5 6}}                    % ⇒ 30
local X in {Mult 2 3 X} {Show X} end  % ⇒ 6

% Erroenous calls: {Mult 5 (99)} {Mult (5) 99}
% The following are eqiuvalent: Infix operators bind strongest!
{Show {Mult 5 99}}
{Show {Mult 2 + 3 9 * 11}}
```

$$\texttt{F = fun \{\$ } X_1 \texttt{ ... } X_n \texttt{\} S end} \approx \texttt{fun \{F } X_1 \texttt{ ... } X_n \texttt{\} S end}$$

- Procedure equality is based on names.
- Mutually declared functions are declared like normal functions.

"Procedure invocation style":

$$\texttt{R = \{F } X_1 \texttt{ ... } X_n \texttt{\}} \approx \texttt{\{F } X_1 \texttt{ ... } X_n \texttt{ R\}}$$

Literals

Literals are symbolic entities that have no internal structure; e.g., `hello`.

- There are also 'names', which are guaranteed to be worldwide unique.
- `{NewName X}` is the only way to create a name and assign it to `X`.
- Names cannot be printed.

```
local X Y B in
   X = foo
   {NewName Y}
   B = true
   {Show [X Y B]}  % ⇒ [foo <OptName> true]
end
```

Records —Hashes & Tuples

A *tuple* is a literal that has data with it —the literal is then referred to as the "label". If `T` is a tuple of n items, then `T.i` is item $i \in 1..n$.

```
declare J

J = jasim('Farm' 12 neato)  % Tuple of three values

{Show J}   % ⇒ jasim('Farm' 12 neato)
{Show J.2} % ⇒ 12
```

A *record* is a tuple where the projections `T.i` are not numbers but are stated explicitly —and called "features". This is also known as a "hash", where the projections are called "keys".

```
declare J = jasim(work: 'Farm' family:12 title: myman)
{Show J} % ⇒ jasim(family:12 title:myman work:'Farm')
{Show J.family} % ⇒ 12
```

This approach is inherited from Prolog.

Tuples are also known as *terms*; everything can be thought of as a term. E.g., we can make trees using terms:

```
declare G = grandparent(dad(child1 child2) uncle(onlycousin) scar)

{Show G.1.1} % ⇒ child1
{Show {Value.'.' G 1}} % ⇒ dad(child1 child2)

% {Show G.nope} % ⇒ Crashes since ‘‘G’’ has no ‘‘nope’’ feature
% {CondSelect R f d X}  ⇒  X = if R has feature f then R.f else d end
local X in {CondSelect G nope 144 X} {Show X} end % ⇒ 144

% {AdjoinAt R f v R'}  ⇒  R' is a copy of R additionally with R'.f = v
% This is an ‘‘update’’ if R.f exists, and otherwise is a new feature.
local H in {AdjoinAt G nope 169 H} {Show H.nope} end % ⇒ 169
```

Remember: Commas are useless!

- ◇ Since everything in Oz is first-class, we have `r.p` ≈ `{Value.'.' r p}`.
- ◇ Here is the library of methods for working with records.
  - ○ Which includes folds on records!
- ◇ `{Arity R X}` assigns `X` the list of features that `R` has.

A standard tuple former name is `'#'`, and it may be used infix by dropping the quotes.

```
{Show 1#2#3}      % ⇒ 1#2#3
{Show '#'(1 2 3)} % ⇒ 1#2#3
{Show '#'()}      % ⇒ '#', empty tuple
{Show '#'(1)}     % ⇒ '#'(1), singleton tuple
```

Likewise, lists are just tuples, which are just records having label `'|'`.

Pattern Matching

Besides projections, `record.feature`, we may decompose a record along its “pattern”.

Below, taking binary trees to have a value and two children, we *declare* three names `Val, L, R` by decomposing the shape of the input `Tree`.

```
declare
fun {GetValue Tree}
   local tree(Val L R) = Tree in Val end
end

{Show {GetValue tree(1 nil nil)}} % ⇒ 1
% {Show {GetValue illFormed}} % ⇒ Crashes: Cannot match tree pattern.
```

We may also perform explicit pattern-matching, which implicitly introduces names.

```
local T = person(jasim farm 12) in
   case T
   of tree(X Y Z) then {Show Y}
   [] person(X Y Z) then {Show X}
   else {Show 'I'm so lost'}
   end end
```

We may omit the `else` and any `[]`-alternative clauses, but may encounter an exception if all matches fail. In which case, we could enclose the dangerous call in `try ··· catch _ then ··· end` to ignore an exception and continue doing something else.

Lists

Oz supports heterogeneous lists.

- ◇ Lists are just tuples —whence projections 1 and 2!

```
% Lists items seperated by a space.
declare L = ['a' 2.8 "3" four]

% Projection functions ‘‘head’’ and ‘‘tail’’
{Show L.1} % ⇒ a
{Show L.2} % ⇒ [2.8 [51] four] ; strings are lists of ascii chars

% Lists are constructed using |, ‘‘cons’’.
{Show 'x'|2|'z'|nil } % ⇒ ['x' 2 'z']

% Decompose L into the ‘‘pattern’’ X|Y|Tail
case L of X|Y|Tail then {Show Y} end   % ⇒ 2

% Lists may also be written in prefix, or ‘record’, form.
{Show '|'(1 '|'(2 nil))} % ⇒ [1 2]
```

```
% Example higher-order function on lists
fun {Map XS F}
   case XS of nil  then nil
           [] X|Xr then {F X}|{Map Xr F} end end

{Show {Map [1 2 3 4] fun {$X} X*X end}} % ⇒  [1 4 9 16]
```

Lazy Evaluation

Demand-driven: Get as much input as needed to make progress.
- ◇ Mark functions using the `lazy` keyword.

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

case {Ints 3} of X|Y|More then {Show X + Y} end  % ⇒  X = 3, Y = 4   ⇒   7
```

'=' is Unification, or 'incremental tell'

Operationally `X = Y` behaves as follows:
1. If either is unbound, assign it to the other one.
2. Otherwise, they are both terms.
   - ◇ Suppose $X \approx f(e_1 \ldots e_n)$ and $Y \approx g(d_1 \ldots d_m)$.
   - ◇ If `f` is different from `g`, or `n` different from `m`, then crash.
   - ◇ Recursively perform $\mathtt{e}_i$ `=` $\mathtt{d}_i$.

"Unification lets us solve equations!"

```
local X Y in
   % Fact: We know that Jasim loves kalthum
   Y = loves(jasim kalthum)
   % Query: Who is loved by Jasim?
   loves(jasim X) = loves(jasim kalthum)
   {Show X} % ⇒  kalthum
end
```

This is why Oz variables are single assignment!

For Boolean equality, one uses `==` or, alternatively, `{Value.'==' X Y R}` to set `R` to be true if `X` ≈ `Y` and false otherwise. Likewise, for other infix relations `\=, =<, <, >=, >` and lazy infix connectives `andthen` and `orthen`.

Here's another example; "wildcard" `_` is used to match anything —so-called "anonymous variable".

```
declare Second L
[a b c] = L
L = [_ Second _]
{Show Second} % ⇒  b
```

Whence, pattern matching is unification!

Unification is the primary method of computation in Prolog.

Control Flow

- ◇ Empty `skip`: Do nothing.
- ◇ Sequencing $\mathtt{S}_1$ $\mathtt{S}_2$: Execute $\mathtt{S}_1$ then $\mathtt{S}_2$.
  - ◦ A single whitespace suffices to sequence two statements.

  ```
  local X Y in skip X = 1 Y = 2 end
  ```
- ◇ Conditional `if B then` $\mathtt{S}_1$ `else` $\mathtt{S}_e$ `end`: Usual conditional if `B` is Boolean; crash otherwise.

  ```
  % Contraction
    if B1 then S1 else if B2 then S2 else S3 end end
  ≈ if B1 then S1 elseif B2 then S2 else S3 end

  % No ‘‘else’’
    if B then S end
  ≈ if B then S else skip end
  ```

  Here's a for-loop for printing the first 10 natural numbers —c.f. `Map` above.

```
local [From To Step DoTheThing] = [0 9 1 Show]
in {For From To Step DoTheThing} end
```

Mutable State

```
declare C

% Create a memory cell with an initial value
C = {NewCell 0}

% Access the value using ‘‘@’’.
{Show  C}   % ⇒ <Cell>
{Show @C}   % ⇒ 0

% Update using ‘‘:=’’.
C := @C + 1
{Show @C}   % ⇒ 1
```

Class A record consisting of method names and attributes.

Object A record consisting of a class and a private function from the class' names to values.

- `obj.method` denotes calling the private function of `obj` with name `method`.

See here for examples.

Reads

- Oz Standard Library
- Oz Demo —a brief & friendly introduction to Oz
- First Steps in Oz
- Tutorial of Oz —slightly outdated but a very useful read
- A review of Oz and its implementation with Mozart —terse & accessible 7 page read
- Logic Programming in Oz with Mozart —explains how to do Prolog-like programming in Oz