The Oz Browser

Konstantin Popov

mozart

Abstract

The Oz Browser is a concurrent output tool for displaying possibly partial information about the values of variables.

Credits

Mozart logo by Christian Lindig

License Agreement

Contents

The Oz Browser

The Oz Browser is a concurrent output tool for displaying possibly partial information about the values of variables.

The Browser is written in Oz itself

Note: However, it uses a number of private features of the Oz implementation, which are not described in this manual or somewhere else, but required to achieve the special Browser functionality, like browsing in local computation spaces. and constitutes a nontrivial example of concurrent programming in Oz. Its source code is shipped with the Oz distribution.

Browser has a graphical user interface which is built using the Oz Tk interface (see *"Window Programming in Mozart"*). Hopefully, it is mainly 'self-explaining' and its use is quite straightforward.

Figure 1.1 The Oz Browser.
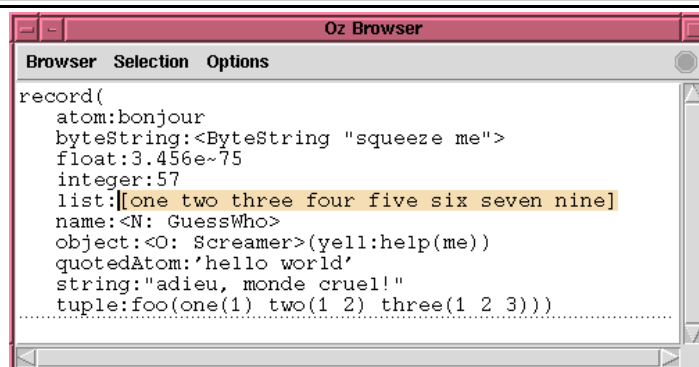
Browser can be easily used in an Oz application as an 'embedded' viewing tool. A Browser widget can be placed in any window and used for browsing of an application's output.

Using the Browser

In this section the functionality of the Browser is presented. Simultaneously, notions required for reading of the following sections are explained.

2.1   Invoking the Browser

The Browser can be invoked by the predefined unary procedure `Browse`. For example,

```
{Browse X}
```

creates a browser window and shows the current constraint on the variable `X`.

A browser window can be closed via the Close entry of the Browser menu, or by the `C-x` keyboard accelerator.

Normally the `Browse` procedure is used in the top-level computation space. Browsing from local computation spaces is not concurrent [1], and there is a number of restrictions of the Browser functionality, which are considered in the Section 2.9.

2.2   The User Interface

A Browser window (Figure 1.1) consists of a text widget in which browsed information is shown, a menubar and scrollbars. The red button on the right side of the menubar, called the *break* button, is used to stop browsing as soon as possible.

Menu entries can be active or passive, depending on the Browser state. On the right side of menu entries their keyboard accelerators are specified, if any. Using an accelerator of a (currently) passive entry has no effect.

The view of the text widget can be adjusted by the scrollbars. The number of lines in the text widget depends on information that is browsed, and is limited only by the Tk implementation. Lines in text widget are never wrapped, and its width is determined automatically by the Tk library.

The Browser automatically changes the layout of information shown in its text widget whenever the window size is changed by the user, or user changes the currently used font.

---

[1]That is, incrementally imposed constraints are not reflected by the Browser. One could say that a browser makes a 'snapshot' of constraints.

The usual 'cut-and-paste' mechanism can be used with the Browser if you want to paste a part of a term's representation somewhere else. A text fragment is selected like in Emacs or the X11 terminal emulator (xterm), except that the `Shift` key must be kept pressed during the selection, and that there are no word- or line- selection modes. On Windows, a selected text is copied by means of `C-q`

2.3 Browser Module

The `Browser` module exports the following values:

`Browser.'class'`

is an Oz class that implements the Browser.

`Browser.object`

is an instance of `Browser.'class'`[2]:

```
BrowserObject = {New Browser.'class' init}
```

`Browser.browse`

is an unary procedure used to tell `Browser.object` to browse its argument. It is also available as procedure `Browse` in the OPI:

```
Browse = proc {$ X} {Browser.object browse(X)} end
```

Methods of the `Browser.'class'` are summarized in the Chapter 4.

2.4 What Is Browsed: A Look Under The Hood

Oz Browser displays the current information about values of variables. This information is stated by means of constraints stored in constraint stores. Constraint that describes a value of a variable can be represented as a directed graph which is shown in textual form.

Due to the properties of the Kernel Oz[3], a constraint in the store can be only a conjunction of the following types of formulae:

- Equations of the form $x = y$, where x and y are distinct variables.

- Equations of the form $x = s$, where s is either a primitive value or a record $l(l_1 : x_1, \ldots, l_n : x_n)$, and $x, x_1, \ldots, x_n$ are variables.

- Disjunctive constraints of the form $x = n_1 \vee \ldots \vee x = n_k$, where x is a variable and $n_1, \ldots, n_k$ are integers which are allowed to be an element of a finite domain.

- Feature constraints of the form $x = l(l_1 : x_1, \ldots, l_n : x_n \ldots)$, where $x, x_1, \ldots, x_n$ are variables.

---

[2]Actually, two *distinct* things are called 'Browser' here: first, this is the tool described in this chapter itself, and, second, this is a particular instance of the tool (a `Browser.'class'` object).

[3]Namely, due to the constraint system itself and restricted use of constraint predicates in the Kernel Oz.

For instance, the value of the variable *X* from the example

```
X = a(1 b(r:2))
```

is described as follows: $x = a(x_1, x_2) \wedge x_1 = 1 \wedge x_2 = b(r : x_3) \wedge x_3 = 2$ Since a store keeps its constraint only up to logical equivalence in the Oz Universe, one can say that for every variable *x* there is at most one binding for *x*, i.e. a formula of the forms mentioned just above.

Additionally, during the elaboration of an expression the Oz System imposes a total order '$\prec$' on all variables, which occur in it[4], so that a constraint can only contain either $x = y$ if $x \prec y$, or $y = x$ otherwise.

Informally, a graph representation of a constraint on a variable *x* emerges when that constraint is traversed up to primitive values or yet unbound variables. Leaf nodes of a graph denote those primitive values and unbound variables, non-leaf nodes denote records, and edges of the graph model occurrences of variables in records.

A graph shown by the Browser is constructed as follows. Let us denote the set of variables to be browsed as $\mathcal{V}$, whereat initially $\mathcal{V}$ contains the variable *x* passed to `Browse`: $\mathcal{V} = \{x\}$; the set of already browsed variables as $\mathcal{P}$ [5], initially $\mathcal{P} = \emptyset$, and the set of couples $\langle x, y \rangle$ as $\mathcal{E}$, initially $\mathcal{E} = \emptyset$. Each node carry an auxiliary label – a variable it represents; couples from $\mathcal{E}$ denote edges between nodes labeled by its variables. Nodes of a graph are created iteratively; at each step a variable *y* is extracted from $\mathcal{V}$, and one of the following rules is applied:

- If the constraint store does not contain any binding for or a constraint on *y*, then a leaf node labeled by *y* is created that represent the variable itself. The variable *y* is added to $\mathcal{P}$.

- If there is an equation of the form $y = z$, then $\{z\} \cap \mathcal{P}$ is adjoined to $\mathcal{V}$, and *y* is replaced by *z* in every couple of $\mathcal{E}$ and among nodes' labels. This step can be seen informally as 'dereferencing'; it serves for minimizing the graph representation of a constraint.

- If there is an equation of the form $y = s$, where *s* is a primitive value, then a leaf node labeled by *y* is created that represent that value. The variable *y* is added to $\mathcal{P}$.

- If there is an equation of the form $y = s$, where *s* is a record $l(l_1 : x_1, \ldots, l_n : x_n)$, then a non-leaf node labeled by *y* is created that represent that record. The variable *y* is added to $\mathcal{P}$. The set $\{x_1, \ldots, x_n\} \cap \mathcal{P}$ is adjoined to $\mathcal{V}$, and *n* new couples $\langle y, x_i \rangle$ are added to $\mathcal{E}$.

- If there is a disjunctive constraint on *y*, then a leaf node labeled by *y* is created that represent the corresponding finite domain. The variable *y* is added to $\mathcal{P}$.

---

[4]Informally, $x^\beta \prec_\alpha y^\gamma$, where α, β and γ are computation spaces, so that $\alpha \preceq \beta$ and $\alpha \preceq \gamma$ (that is, α is below or equal both β and γ), if $\beta \preceq \gamma$ and either *(i)* $\alpha \neq \gamma$, *(ii)* *x* is an unconstrained variable while *y* is a constrained one, or *(iii)* *y* was created earlier than *x*. Otherwise, an *arbitrary* order is chosen by the Oz Engine. Note that the order on local to a computation space variables changes unpredictably during the garbage collection.

[5]The set $\mathcal{P}$ is needed to handle constraints over rational trees, which allow graphs described here to be cyclic.

- If there is an feature constraint of the form $y = l(l_1 : x_1, \ldots, l_n : x_n \ldots)$, then a non-leaf node labeled by y is created that represent that partially known record. The variable y is added to $\mathcal{P}$. The set $\{x_1, \ldots, x_n\} \cap \mathcal{P}$ is adjoined to $\mathcal{V}$, and n new couples $\langle y, x_i \rangle$ are added to $\mathcal{E}$.

Construction of the graph is stopped whet the set $\mathcal{V}$ becomes empty. It is continued automatically when additional constraints are added to the store.

In the scope of this documentation, the textual representation of a (sub)graph is called (sub)term[6]. A (sub)term can be primitive or compound, similarly to the Oz values.

Numbers and records are shown as one would expect. Procedures, cells, chunks, spaces and threads are shown by their print names. Finite domains have a special representation: for instance, if an FD variable is constrained to the domain $\{1,2,3,7,8,9\}$, then it is shown as `_{ 1..3 7..9 }`. Partially known records are represented as records, but with additional ellipses just before the closing parentheses: `ofs(a: 1 b: 2 ...)`.

Additionally, the following Oz data types are treated specially:

Lists   have two representations – a sequence of cons cells (like `1|2|A`) and the representation of well-formed lists in square brackets (like `[1 2 3]`). The second representation is choosen whenever a list can be decided to be well-formed when only N its elements are considered. N is double of the current value of the `Width` browse limit, described in Section 2.5.

Chunks   that constructed from non-primitive values (like classes and objects) are represented as compound terms if the `Chunks` representation detail option is set, and if this option is not set then the string `(?)` is added to the representation.

Strings, Virtual Strings   are shown in readable form, i.e. by enclosed in quotes ASCII strings, if the corresponding `Strings` or `Virtual strings` option, repectively, is set.

Note that browsing of virtual strings is not monotonic. For instance, constraints on the variable `S`

```
declare S in S = _#[101 108 108 111]
```

is browsed in this case as `_#ello` [7]. If the additional Oz line

```
S = [72]#_
```

is feed, the Browser changes the term's respresentation to `H#ello` and *not* to `Hello`; you have to issue the `Rebrowse` command to obtain 'Hello'.

An extended format of print names of variables, procedures, cell, chunks, spaces and threads can be requested by setting the `Variable Status` and `Names And Procedures` options.

---

[6]Browser terms are similar to Oz (syntactic) terms as defined in *"The Oz Notation"*, but they have different origins.

[7]The string `[101 108 108 111]` is a string of the ascii codes for the characters *e*, *l*, *l* and *o* respectively.

2.5 The Basic Output Format

This section describes the simplest Oz value's representation the Browser supports. In this case the Browser thinks of a value as of a tree that leaf nodes are primitive values and non-leaf nodes are records. Note that equal subtrees are represented as many times as they occur; this is a simplification with respect to the value graph described in Section 2.4. More concise representations are discussed in Section 2.7.

The description is divided into the following issues:

- Browsing of arbitrarily deep and wide terms.

- Using parenthesis to override the term constructor precedences.

- Printing layout for compound terms.

Each (sub)term is shown in a certain viewing form, namely, it can be drawn completely, partially or it can be shrunken:

Completely   means that the term is shown if it represents a primitive value, finite domain or a variable; or all its subterm(s) are shown in some form if the term is compound.

Partially   means that only the N first subterms of the compound term are shown, the rest of them is replaced with '„,';

Shrunken   means that the term is shown in the form '„,'.

The form of each (sub)term is determined by values of the width and depth browse limits:

Width   specifies for each compound (sub)term the number of its subterms that are shown, i.e. the number N from above.

Depth   specifies the depth in a term's representation where subterms are shrunken.

These parameters can be set via the `Display Parameters` submenu of the `Options` menu. Note that increasing of values `Depth` and `Width` affects all currently shown terms.

Terms shown by the Browser are built with respect to the Oz term constructor precedences as defined in Oz Notation , and uses round parentheses to override them.

The Browser tries to produce the most compact printing layout for compound terms. Namely, it packs in one row as many subterms as possible, while usual minimal subterm indentation is provided. The only exception of this rule is that record subterms are shown in one column if they don't fit in one row. However, this can be switched off via turning off the option `Record Fields Aligned` of the `Layout` submenu of the `Options` menu.

A non-shrunken (sub)term can be shrunken explicitly by means of the `Shrink` commands of the `Selection` menu. The command is applied to a selection, which is described in Section 2.8. A partially shown or shrunken (sub)term can be expanded:

- If a (sub)term was shrunken, it is replaced by a subterm which is `Depth` deep, where `Depth` is the Browser's current expansion limit. The expansion limits can be set by means of the `Display Parameters` submenu of the `Options` menu.

- If a (sub)term was partially shown, its further `Width` subterms are viewed, where `Width` is the Browser's expansion limit.

2.6 Controlling Browsing

Browser draws terms sequentially, in the depth-first, left-to-right fashion, one term after each other. Drawing process can be stopped nearly all the time by pressing the *break* button (or by using the `Break` entry of the `Browser` menu). When browsing is being stopped, (sub)term(s) yet to be drawn appear in shrunken form, and started but not yet completed (sub)term(s) appear partially.

More than term can be shown simultaneously in a Browser window. The Browser contains a first-in-first-out buffer where information about browsed terms is kept. The buffer size limits the number of simultaneously shown terms; it can be set via the `Buffer` submenu of the `Options` menu. All terms in the buffer can be removed via the `Clear` entry of the `Browser` menu. All terms except the last one can be removed via the `Clear All But Last` entry.

2.7 Represntation of Rational Trees

The Browser can represent a cyclic tree, i.e. a tree-like graph that contains 'backward' edges (edges starting in a node below its target). A cyclic tree representation is built as follows:

- A subterm representing a target node obtains a unique prefix of the form C*N*=, where *N* is a number.

- For all backward to that node edges pseudo-nodes are created, which are represented as C*N*.

For example, the value of *X* determined by the expression

```
declare
N = {NewName}
X = a(f1:X f2:N f3:N)
in {Browse X}
```

is browsed as `C1=a(f1:C1 f2:<N: 'N'> f3:<N: 'N'>)` .

A textual representation of a value graph described in Section 2.4 is generated using the 'minimal graph' representation option. Informally speaking, in this mode equal subgraphs are represented only once; all subsequent subgraphs that are equal to an already created one [8] are represented as a reference to it. For instance, the Browser output for the example above would be `R1=a(f1:R1 f2:R2=<N: 'N'> f3:R2)` .

---

[8]Two graphs are equal here iff they are unifiable.

The representation options are set by means of the `Representation` submenu of the `Options` menu, by choosing one of the `Tree`, `Graph` and `Minimal Graph`.

Note that additional parenthesis are inserted around textual representation of (sub)terms with the prefixes R*N*= or C*N*=, if necessary. For instance, the Browser output of the example:

```
declare
X = a(1|2|3|b(c) b(c))
in {Browse X}
```

is `a(1|2|3|(R1=b(c)) R1)`.

The only operation defined on reference names is dereferencing. Its effect is scrolling to a term tagged with the reference name, and selecting it. The `Deref` command can be invoked from the `Selection` menu or by the keyboard accelerator, similarly to `Expand` and `Shrink` commands.

2.8 Actions

An arbitrary shown (sub)term can be selected. This is achieved by clicking the left mouse button. Deselection proceeds by clicking the right one. Selected (sub)terms are shown in the text widget on a wheat colored background. Selected (sub)terms are targets for both pre-defined browser operations (like `Shrink` and `Expand` described in SectionSection 2.5) and user-defined actions.

The *Actions* mechanism provide for application of unary procedures with the current selection as an argument. There are two predefined actions: `Show` and `Browse`. There can be user-defined actions, which are added by means of the `add`, `set` and `delete` methods of `BrowserClass`:

```
%%
declare
P = proc {$ S} {Browse {Value.status S}} end
in
{Browser add(P label:'Show Status')}
{Browser set(P)}
%% use the action now!
{Browser delete(P)}   % delete(all)
{Browser set(Show)}
```

In this example a new action `P` is created (which effect is printing the status of a selection), then it is added (`add`) and set (`set`) as the current one. At this point user can either click the middle mouse button or use the `Apply Action` entry of the `Selection` menu in order to invoke it. After that the action is removed, and the predefined `Show` is set as the current.

A useful application of this mechanism is the action which equates two subsequent selections:

```
declare
class UnifyTwoClass from Object.base
   prop final locking
   attr first: _ state: start
   meth click(S)
      case @state == start then
         {Show 'First...'}
         first := S
         state := continue
      else
         {Show 'Unify!'}
         S = @first
         state := start
      end
   end
end
UnifyTwo = {New UnifyTwoClass noop}
proc {UnifyAction S}
   {UnifyTwo click(S)}
end
in skip
```

Now we can set this action and browse, for instance, a list constructor with two variables:

```
{Browser createWindow}
{Browser add(UnifyAction)}
{Browser set(UnifyAction)}
{Browser option(representation mode:graph)}
%%
{Browse _|_}
```

If we will click the middle mouse button subsequently on the list constructor and the second variable, we will get a cyclic list: `C1=_|C1`.

2.9 Browsing in Local Computation Spaces

There are the following distinguishing features of using the Browser in a computation space:

- Browsing in a computation space *is not* concurrent.

- Print names of variables, (Oz) names, procedures, cells, chunks, spaces and threads are quoted. For instance, a variable `A` will be browsed as `'A'`, and not as `A`.

- Print names of mentioned above objects are always generated in the extended format, without any respect to the current value of the options `Variable Status` and `Names And Procedures`.

- Feature constraints are converted to records, that is, their representation does not contain ellipses just before the closing parentheses.

- Generally, builiding `graph` and `minimal graph` representation yields *wrong* results. For instance, two equal variables are not detected as being equal; whereas two tuples `a(_)` and `a(_)` are detected as being equal.

- The Browser needs much more memory and runtime to perform browsing, in particular when large lists are browsed.

- Continuous browsing of big different terms leads to significant memory leakage in the Oz Emulator, which is limited by the total size of a textual representation of all browsed terms.

The differences mentioned above are straightforward once there is the understanding how the Browser works in that case.

The point here is that the Browser operates in the top-level computation space, whereas a constraint describing a value of a given variable can contain variables, names, procedures and so on that belong to the local computation space. These variables cannot occur in constraints in the top-level one; therefore, each constraint browsed in a local computation space is converted to a constraint which *does not contain* any variables, names and so on this transformation, called here *reflection*, takes place in the local computation space, where `Browse` call was issued. After that the reflected constraint is passed to the top-level computation space by a special builtin, and actually browsed.

The reflection step can take place only once for a browsed constraint; that is, browsing is not concurrent in this case.

Objects of mentioned above types are replaced by atoms during reflection. The number of new atoms created this way is limited by the size of a constraint graph to be browsed. Since atoms are not a subject for garbage collection in the Oz System, the memory consumed by the System grows up.

Since the reflection process must terminate, the Browser builds a minimal graph representation of a constraint, which is passed to the top-level computation space. This explains the computational complexity of browsing in local computation spaces.

2.10 Setting Options

Every Browser option that can be set by means of the `Options` menu can be set also by means of the `option` method of the `BrowserClass`. Its format is

```
option(group option: value)
```

There is the complete list of possible options with example values:

```
option(buffer size:1)
option(buffer separateBufferEntries:false)
option(representation mode:minGraph)
option(representation detailedChunks:true)
```

```
option(representation detailedNamesAndProcedures:false)
option(representation detailedVarStatus:false)
option(representation strings:true)
option(representation virtualStrings:false)
option(display depth:100)
option(display width:2000)
option(display depthInc:2)
option(display widthInc:5)
option(layout size:14)
option(layout bold:false)
option(layout alignRecordFields:true)
```

There is an additional group special which allows to control the geometry of the Browser window:

```
option(special xSize:800)
option(special ySize:600)
option(special xMinSize:300)
option(special yMinSize:200)
```

Browser Commands and Options

The meanings of the menu entries of the Browser are summarized in this section.

3.1 The Browser Menu

About...

Displays a small dialog containing information about the product.

Break	C-c

Stops browsing as soon as possible.

Deselect

Deselects the current selection, if any.

Clear	C-u

Removes all browsed information from the Browser window, if any.

Clear All But Last	C-w

Removes all browsed information from the Browser window except the last term, if any.

Refine Layout	C-l

Checks the current term layout and optimizes it whenever possible.

Close	C-x

Closes the Browser.

3.2 The Selection Menu

Expand	e

Replaces the selected shrunken (sub)term „ , by a subterm(s), or views further subterm(s) of a partially shown selected (sub)term.

Shrink	**s**	

Shrinks the selected (sub)term.

Deref	**d**	

If a reference (R*N* or C*N*) subterm is selected, the Browser exposes and selects the referenced (sub)term.

Rebrowse	**C-b**

Removes and creates again a selected (sub)term.

SetAction	

Shows a submemu an action can be choosen from.

Apply Action	**C-p**	

Applies the currently set action to the current selection, if any.

3.3 The Options Menu

Buffer...	

Creates a dialog used for setting the buffer size. and enabling drawing a graphical separator between shown terms.
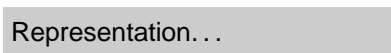
Representation...	

Creates a dialog used for setting the representation mode (building `Tree`, `Graph` or `Minimal Graph`), detail level (for `Chunks`, `Variable Status` and `Names And Procedures`), and type (displaying `Strings` or `Virtual Strings` as such).

Display Parameters...	

Creates a dialog used for setting the browse limits (`Depth` and `Width`) and the expansion increments (also `Depth` and `Width`).

Layout...	

Creates a dialog used for setting a font to be used (`Font Size` and whether it should be `Bold`), as well as setting Browser to show record subterms in one column if they don't fit in one row (that is, to `Align Record Fields`).

Using Browser Objects

All Browsers in the Oz system are Oz objects. The `BrowserClass` is the class defini-
tion of these objects. In this section the usage of these objects is described.

4.1 Creating A Browser Object

One can create Browser objects similarly to the pre-defined `Browser` one:

```
BrowserObject = {New Browser.'class' init}
```

The plain `init` method makes a newly created Browser placed on its own toplevel
widget. One can tell a Browser object to use a Tk frame (see for details) as its toplevel
frame:

```
declare W F BrowserObject in
%%
W = {New Tk.toplevel tkInit(bg:ivory)}
{Tk.send wm(geometry W "500x300")}
%%
F = {New Tk.frame tkInit(parent : W
                         bd     : 3
                         bg     : white
                         relief : groove
                         width  : 450
                         height : 250)}
{Tk.send pack(F fill:both padx:10 pady:10 expand:true)}
%%
BrowserObject = {New Browser.'class' init(origWindow: F)}
{BrowserObject createWindow}
%%
{BrowserObject
 browse(
    record(
       atom: bonjour
       quotedAtom: 'hello world'
       list: [one two three four five six seven nine]
```

```
integer: 57
float: 34.56e~76
string:"adieu, monde cruel!"
byteString:{ByteString.make "squeeze me"}
tuple: foo(one(1) two(1 2) three(1 2 3)))))}
```

The window produced that way is shown on Figure 4.1. Such a browser is called an embedded Browser, compared to a stand-alone Browser using its own toplevel widget.

Note that when a Browser object is created, either stand-alone or embedded, its

Figure 4.1 An Embedded Application Browser
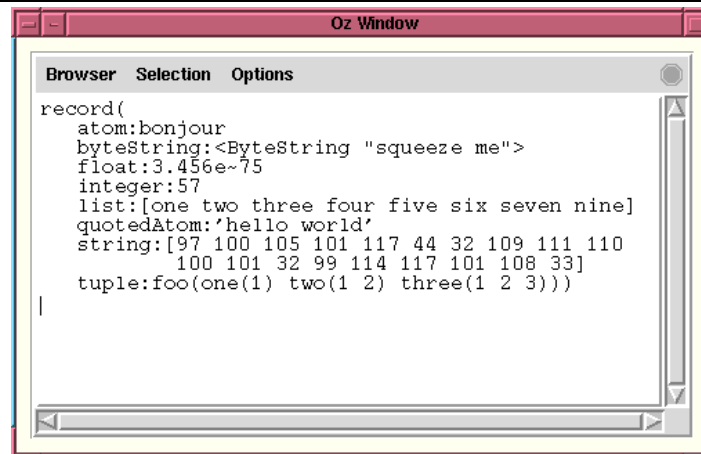
window does not appear immediately. This happens when either `createWindow` or `browse` methods are applied.

A Browser window can be closed by means of the `closeWindow` method. Each time a Browser window is closed its buffer is flushed. A Browser object itself can be closed by means of the `close` method, which deletes its window and frees used by the object memory. A subsequent application of a Browser object closed this way proceeds as it were just created\footnote{This feature is used in the pre-defined `Browser`, which can be closed many times but never dies. Note that this behavior *is not* 're-creating' since no new Browser object is created immediately.}.

Note that the `closeWindow` method has no effect for embedded application browsers.

An embedded browser and its frame should be closed by sending the `close` messages on the Browser object and on the frame object in that sequence. For instance, in our example it would be

```
{BO close}
{F close}
```

4.2 Browser Object Methods

Browser objects are controlled by user through the Browser GUI, and by means of application of them to the `BrowserClass` methods. At the glance, there are the following

public methods of a Browser object, where x and y are certain internal default values:

```
meth init(origWindow: OrigWindow <= X)
meth createWindow
meth closeWindow
meth close
meth browse(Term)
meth break
meth clear
meth clearAllButLast
meth rebrowse
meth option(...)
meth add(Action label:Label <= Y)
meth set(Action)
meth delete(Action)
meth refineLayout
```

Index