The PicoLisp 'edit' Function
Browsing a Distributed Database With a Text Editor

Alexander Burger
abu@software-lab.de

2011-07-29

Abstract

The 'edit' function in PicoLisp can edit the value and properties of any Lisp symbol, and follow references to symbols referred from these data. As database objects (including those of remote/distributed systems) are nothing more than Lisp symbols (mapped transparently into the current scope), a single edit session can possibly span the whole system.

Using 'edit'

On Browsing the Database or Arbitrary Data Structures and Definitions

A quite powerful - but little known - function in PicoLisp is 'edit[1]'.

It allows you to edit any Lisp symbol (intern, transient or extern), by pretty[2]-printing its value and properties to a temporary file, calling an external editor, and 'read[3]'ing back the changes when done. For the external editor, currently 'vim' is supported.

The clou: You can "click" on any other symbol somewhere embedded in the nested structures of the value or property list, to have it added to the editor screen, and thus browse through potentially the whole system.

This works transparently not only for internal symbols, but also for transient (which are normally not directly accessible) and external (database) symbols. In the case of external symbols, it doesn't even matter whether these are objects in a local database, or whether they reside on remote machines in a distributed system (except that remote objects cannot be modified).

PicoLisp Symbols

A symbol in PicoLisp consists of three - possibly empty - components: A value, a property list and a name. Of those, usually only the value and the properties are modified during programming. The value is nothing more than a special property, used implicitly for prominent purposes like variable bindings or function definitions.

There exist a large number of functions to access, set or modify a symbol's value or property list. Setting the value and a few properties of the symbol `X`:

```
: (setq X "Hello")
-> "Hello"

: (put 'X 'a 1)
```

[1] http://software-lab.de/doc/refE.html#edit
[2] http://software-lab.de/doc/refP.html#pretty
[3] http://software-lab.de/doc/refR.html#read

```
-> 1

: (with 'X
   (=: b 2)
   (push (:: lst) "OK" '(a b c d) 17) )
-> 17
```

These data can be accessed individually

```
: X
-> "Hello"

: (val 'X)
-> "Hello"

: (get 'X 'lst)
-> (17 (a b c d) "OK")
```

or looked at as a whole, using the function 'show[4]':

```
: (show 'X)
X "Hello"
   lst (17 (a b c d) "OK")
   b 2
   a 1
-> X
```

`show` displays the symbol's name (here `"X"`), then the value (here `"Hello"`) on the same line, followed by all properties, each on its own line indented by three spaces.

Editing a Symbol

Instead of just `show`ing the symbol to the console, you can use `edit` to get a similar display in a 'vim' session:

```
: (edit 'X)
```

The editor's window will appear as:

```
X "Hello"
   a 1
   b 2
   lst (17 (a b c d) "OK")

(********)
```

The difference to a plain `show` is that you can change the value or properties.

The pattern (********) is used by `edit` internally as a delimiter, and should not be modified.

For example, change the `e` in the value "Hello" to `a`, and the `a` in the `lst` property to `x`:

```
X "Hallo"
   a 1
```

[4] http://software-lab.de/doc/refS.html#show

```
   b 2
   lst (17 (x b c d) "OK")

(********)
```

then exit 'vim' in the normal way, e.g. with ":x". On your console you see

```
: (edit 'X)
# X redefined
# X lst redefined
-> NIL
```

You can use `show` or other commands to see that `X` was indeed changed

```
: X
-> "Hallo"

: (get 'X 'lst)
-> (17 (x b c d) "OK")
```

Browsing

Now let's try the "browsing" capability mentioned above. When `edit` starts up the 'vim' editor, it defines two key mappings for that edit session:

- Once you edit a symbol, you can move the cursor to the first character of some other symbol appearing in the value or properties, and press an upper-case 'K'. This will cause that symbol to be added to the edit session, separated by another (********)
- Pressing an upper-case 'Q' goes one step back to the previous view

We can try this while editing `X`. Moving the cursor to the `x` in the `lst` property, and hitting 'K' gives:

```
x NIL

(********)

X "Hallo"
   a 1
   b 2
   lst (17 (x b c d) "OK")

(********)
```

Now we see both `x` and `X` being edited. Unfortunately, `x` is not very interesting here, as it has only the default value of `NIL` and no properties.

The same effect can be achieved by calling

```
(edit 'x 'X)
```

You can pass any number of symbols to `edit`.

A little more happens if we move down to `lst` again, and hit 'K' on the symbol `d`:

```
d (NIL (let *Dbg NIL (dbg ^)))
   *Dbg ((216 . "/usr/lib/picolisp/lib/debug.l"))

(********)

x NIL

(********)

X "Hallo"
   a 1
   b 2
   lst (17 (x b c d) "OK")

(********)
```

Indeed, now we found something! This is not surprising, though, as 'd[5]' has a definition in the debugger context. The value is the function

```
(() (let *Dbg NIL (dbg ^)))
```

and the '*Dbg[6]' property contains the file and line number of its source.

Transient Symbols

We can use `edit` to inspect itself.

```
: (edit 'edit)
```

The result looks meager

```
edit (@
   (let *Dbg NIL
      (setq "*F" (tmp '"edit.l"))
      (catch NIL ("edit" (rest))) ) )
   *Dbg ((6 . "lib/edit.l"))

(********)
```

because - as can be seen in the fourth line - `edit` is a short function wich calls `"edit"` (defined in a transient symbol) to do the actual work.

The transient symbol `"edit"` is not directly reachable. In the REPL

```
: (pp '"edit")
(de "edit" . "edit")
-> "edit"
```

we see just the string "edit".

But if we edit `edit`, place the cursor on the first double quote character of "edit" in line four, and press 'K', we get

[5] http://software-lab.de/doc/refD.html#d
[6] http://software-lab.de/doc/refD.html#*Dbg

```
"edit" (("Lst")
   (let "N" 1
      (loop
         (out "*F"
            (setq
               "*Lst" (make
                  (for "S" "Lst"
                     ("loc" (printsp "S"))
                     ("loc" (val "S"))
...

(********)

edit (@
   (let *Dbg NIL
      (setq "*F" (tmp '"edit.l"))
      (catch NIL ("edit" (rest))) ) )
   *Dbg ((6 . "lib/edit.l"))

(********)
```

BTW, you can see another transient function in line 8: `"loc"`. You may click on that one to see its definition.

In contrast, if you look at the 'locale[7]' function

```
(edit 'locale)
```

you'll find in there another, completely different, `"loc"` function. This is an example for the locality of transient symbols. The two `"loc"`'s have nothing to do with each other, and don't conflict in their definitions, yet you can see - and possibly change - them both (separately) in the editor.

Browsing the Database

For the following examples we use the demo application[8] in the PicoLisp distribution. Start it as described in the Getting Started[9] section:

```
$ ln -s /usr/share/picolisp/app
$ pil app/main.l -main -go +
```

Then connect with a browser to 'http://localhost:8080[10]' to get a PicoLisp REPL prompt in your terminal window. Log in as "admin" / "admin" in the browser GUI.

Now you can navigate through the whole database. Start at an arbitrary object. For a first overview, the '*DB[11]' root object is just fine.

```
(edit *DB)
```

You see the external symbol {1}, pointing to the base objects of the entity classes.

[7] http://software-lab.de/doc/refL.html#locale
[8] http://software-lab.de/doc/app.html#minApp
[9] http://software-lab.de/doc/app.html#getStarted
[10] http://localhost:8080
[11] http://software-lab.de/doc/refD.html#*DB

```
{1} NIL
   +Role {3}
   +User {7}
   +Sal {16}
   +CuSu {31}
   +Item {32}
   +Ord {33}
   +Pos {34}

(********)
```

The first one, {3}, is the base of the +Role entity. Move to the opening brace and press 'K'.

```
{3} NIL
   nm (3 . {D1})

(********)

{1} NIL
   +Role {3}
   +User {7}
   +Sal {16}
   +CuSu {31}
   +Item {32}
   +Ord {33}
   +Pos {34}

(********)
```

We see that {3} contains only a single index, the nm (name) property of roles. The number 3 tells us that this index tree has three nodes, and its root node is {D1}.

If we inspect that index root node, by clicking on {D1}

```
{D1} (NIL ("Accounting" NIL . {4}) ("Administration" NIL . {2}) ("Assistance" NIL . {5}))
...
```

The first role in that list is "Accounting", the object {4}.

{4} in turn leads us to

```
{4} (+Role)
   nm "Accounting"
   usr ({12} {11} {10})
   perm (Customer Item Order Report Delete)
...
```

We see an object of class +Role (as expected), with the name "Accounting", the users in the usr list, and a list of permissions.

Again, we might click on the first user, {12}

```
{12} (+User)
   role {4}  # (+Role)
   nam "Sandra Bullock"
   nm "sandy"
   pw "sandy"
...
```

The `role` of that user points back to {4}, as we have a `+Joint` - a bi-directional relation. We might verify this, by exiting `edit` with ":q" and call `(vi '+User)` to inspect the sources

```
...
### Role ###
(class +Role +Entity)

(rel nm (+Need +Key +String))          # Role name
(rel perm (+List +Symbol))             # Permission list
(rel usr (+List +Joint) role (+User))  # Associated users


### User ###
(class +User +Entity)

(rel nm (+Need +Key +String))          # User name
(rel pw (+String))                     # Password
(rel role (+Joint) usr (+Role))        # User role
...
```

showing that `role` of `+User` points to a `+Role` object, and the `usr` property of `+Role` has a list of `+User` objects.

A similar information can also be obtained directly from the runtime system. Go back to the user {4} again

```
(edit '{4})
```

then click on the first character of `+Role` (i.e. the '+' character) in the classes list of {4}.

```
+Role ((url> (Tab) (and (may RoleAdmin) (list "app/role.l" '*ID This)))
   +Entity )
   nm $53165764545663  # (+Need +Key +String)
   perm $53165764545716  # (+List +Symbol)
   usr $53165764545754  # (+List +Joint)
   Dbf (1 . 512)
   *Dbg ((39 . "lib/adm.l")
      (url> 26 . "app/er.l")
      (usr 43 . "lib/adm.l")
      (perm 42 . "lib/adm.l")
      (nm 41 . "lib/adm.l") )
...
```

Note that now we are no longer in a database object, but in the class definition. It shows that `+Role` defines a single method `url>`, is a subclass of `+Entity`, and has relations `nm`, `perm` and `usr`. The property `Dbf` used for database maintenance, and `*Dbg` holds debug information.

You may experiment more. You can click on the '$' of a relation maintenance daemon object, and even on a commented symbol like `+List` or `+Joint`.

Debugging

`edit` comes in handy also during debugging.

You can easily do on-the-fly changes to a function, like inserting a call to print a 'msg[12]', or setting some explicit breakpoint with '![13]', without actually touching the source code.

[12] http://software-lab.de/doc/refM.html#msg
[13] http://software-lab.de/doc/ref_.html#!

To edit a certain object in a large database, it is often easier to find it by going to that object in the GUI. In the demo app, click on the "Orders" menu item to the left, then on the '@' link in the leftmost column of the first order. You should get a form with that order.

Now, in the REPL, you can access the form that is currently shown in the browser via the `*Top` global variable. You may look at it with `(show *Top)`, or edit it with `(edit *Top)`.

You get an awful lot of data, mostly for the GUI components in that form. As before, you can click on any of them to see what they contain.

Scrolling down a bit, there is an `obj` property. This is the database object held by that form.

```
   ...
   evt 0
   obj {B7}  # (+Ord)
   gui ($53165764713535
      $53165764713635
      $53165764713747
   ...
```

Here, it is {B7}. Again, you can click on that,

```
   {B7} (+Ord)
      nr 1
      dat 733027  # 2007-02-14
      cus {C3}  # (+CuSu)
      pos ({A1} {A2} {A3})
   ...
```

and again you are "in" the database. You can follow the links to the customer (the `+CuSu` object {C3}), or the three positions in that order `pos`.

Let's pick the first position {A1}

```
   {A1} (+Pos)
      cnt 1
      pr 29900
      itm {B1}  # (+Item)
      ord {B7}  # (+Ord)
   ...
```

and see a link to the item {B1}, and back to the order {B7}.

The item {B1} leads us to

```
{B1} (+Item)
   nr 1
   inv 100
   pr 29900
   sup {C1}  # (+CuSu)
   nm "Main Part"
   ...
```

in turn pointing to `sup`, the item's supplier {C1}, and so on.

The database objects can be modified here in any conceivable way, but you should be very sure about what you do, if you don't want an inconsistent database. Relations involving index trees or "joint"ed objects need corresponding changes in other objects, and are better avoided. In any case, a change to a DB object will only be manifest if you enter (commit) after exiting from the editor.

Distributed Database

Though the demo app doesn't really make use of remote objects, it contains a hook to experiment with them. If the demo application was started as above, it automatically also listens on port 4040 for remote requests.

A distributed database requires some setup and administration. We don't go into the details here, but a simple setup can be made by starting (in addition to the `app` server above) a stand-alone PicoLisp interpreter in another terminal window

```
$ pil +
```

and initialize the `*Ext` variable as described in 'remote/2[14]'

```
(setq *Ext
   (mapcar
      '((@Host @Ext)
         (let Sock NIL
            (cons @Ext
               (curry (@Host @Ext Sock) (Obj)
                  (when (or Sock (setq Sock (connect @Host 4040)))
                     (ext @Ext
                        (out Sock (pr (cons 'qsym Obj)))
                        (prog1 (in Sock (rd))
                           (unless @
                              (close Sock)
                              (off Sock) ) ) ) ) ) ) ) )
      '("localhost")
      '(20) ) )
```

to let the system know where where to fetch remote objects from.

If you started the remote server on another machine (you didn't forget to open port 4040 in the firewall, did you?), supply its name or IP address instead of "localhost".

Then request the order with the number 1, and `edit` it:

```
: (let Sock (connect "localhost" 4040)
   (ext 20
      (out Sock (pr '(pr (db 'nr '+Ord 1))))
      (prog1 (in Sock (rd)) (close Sock)) ) )
-> {AF7}

: (edit @)
```

From here on, continue as with the local database. Just that the (now remote) order object {B7} appears locally as {AF7}.

```
{AF7} (+Ord)
   nr 1
   dat 733027  # 2007-02-14
   cus {AG3}  # (+CuSu)
   pos ({AE1} {AE2} {AE3})
```

The same holds for the customer and the positions. Clicking on the first position {AE1} gives

[14] http://software-lab.de/doc/refR.html#remote/2

```
{AE1} (+Pos)
   cnt 1
   pr 29900
   itm {AF1}  # (+Item)
   ord {AF7}  # (+Ord)
...
```

Except for the fact that the names of all external symbols appear with an offset, everything else behaves like in the local case.