

Re: the necessity of Lisp's Objects?

Source: <http://coding.derkeiler.com/Archive/Lisp/comp.lang.lisp/2008-01/msg01357.html>

- *From:* Rainer Joswig <joswig@xxxxxxx>
 - *Date:* Tue, 22 Jan 2008 14:26:05 +0100
-

In article

<4b91e167-0570-4f38-abd1-c5b5bdfb56ce@xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx>, Xah Lee <xah@xxxxxxxxxx> wrote:

....

3. The syntax is a bit quirky. In particular, a mathematica programmer sees that sometimes list is written as \$B!H (B(list a b c) \$B!I (B, but sometimes there's this oddity \$B!H (B'(a b c) \$B!I (B (which is syntactically equivalent to \$B!H (B(quote a b c) \$B!I (B

No, it is not.

```
CL-USER 1 > (read-from-string "(a b c)")
(QUOTE (A B C))
8
```

....

I was trying to write a emacs major mode and also writing a tutorial about it, in the past week. In this process, i find it quite hard. Writing a complete, robust, major mode for public consumption will require knowledge of:

- * mode hooks (basic concept)
- * font lock system (some detail)
- * font and faces infrastructure (basic concept)
- * buffer local variables (datil)
- * regex (detail)
- * keymaps (detail)
- * syntax table (detail)
- * abbreviation system (basics)
- * Emacs mode conventions (a lot elisp coding experience)

(the list gives a rough idea, not meant to be complete or exact in some way)

Re: the necessity of Lisp's Objects?

Right. There is quite a lot of to learn to write a good mode. For some modes there seem to be lots of people involved and/or some experts spend significant time improving it.

Mathematica, being high-level as it is, essentially merged the concept of read-syntax, print-syntax, and objects, into just one concept of `$B!H` (BWhat you see is what you get `$B!I` (B_Expressions_ (i.e. the code programmer types). That is, the code is the object, the object is the code. In other words, there is no such `$B!H` (Binternal `$B!I` (B concepts as objects. What you type is the code, as well as the meaning.

For example, let's say a list:

In lisp:
`(list 1 2 3)`

Mathematica:
`List[1,2,3]`

In lisp, we say that the text `$B!H` (B(list 1 2 3) `$B!I` (B is the read syntax for the lisp's list object.

No. `(LIST 1 2 3)` is a textual representation of a function call to `LIST` with data `1 2 3`. The list itself is just written as `(1 2 3)`. Lists have a data syntax that can be printed and read. Like arrays, strings, vectors, numbers, characters, ...

So, the lisp interpreter reads this text, and turns it into a list object. When the lisp system prints a list object, it prints `$B!H` (B(list 1 2 3) `$B!I` (B too, since list object's read syntax and print syntax is the same.

```
? (defun test ()
(print '(1 2 3))
(print (list 1 2 3))
'done)
TEST
? (test)

(1 2 3)
(1 2 3)
DONE
```

Re: the necessity of Lisp's Objects?

In Mathematica, there's no such steps of concept. The list
\$B!H (BList[1,2,3] \$B!I (B is just a list as is. There is no such intermediate
concept of objects.

What are two lists then internally if you read them?
Define two variables. Set both to some lists. What are these lists now?
Say you pass one of these lists to a function.
Does the function get a copy? The same? Don't know?

If you print the Mathematica list and then read it back.
What do you get? A new list (Why?)? The old list? Don't know?
What with symbols. Print a symbol. Read the symbol?
Is it a different symbol? The same? Don't know?

We talk of 'objects' being the data in a running (Lisp) system.
See below. Objects, because we don't see chunks of bits.

Everything in Mathematica is just a \$B!H (Bexpression \$B!I (B,
meaning the textual code you type.

Even Mathematica has the idea of data 'objects':
symbols, numbers, lists, strings, expressions, ...
The documentation of Mathematica is a huge mess when
it comes to describing its basic data types.
Mathematica might be less precisely described compared
to Common Lisp. Common Lisp has a spec (with its
own problems), which allows us to have different and
independent implementations of the programming language
Common Lisp. Mathematica is not described in a way
that one can 'easily' implement the thing independently.
Plus Mathematica is focused on symbol computation
with lots of stuff added to it somehow. For a Lisp user
it is similar to an inhouse closed implementation of a
hacked special purpose Lisp dialect. Common Lisp is a
general purpose standardized programming language
which lays down in the standard how a lot of things
should work (printing, reading, interpreter, compiler,
data types, ...). I'm not saying that it is necessary
beautiful, but the Common Lisp describes the language
in much more detail with more basic facilities.

This is a high-level beauty of Mathematica.

With an implementation that is partly in some low-level language.

Re: the necessity of Lisp's Objects?

Re: the necessity of Lisp's Objects?

You see the beauty of the language. The Lisp programmer sees the details and the implementation, too.

Part of it comes from its 'narrow' main domain: symbolic mathematics (with lots of stuff added somehow). Lisp has a different purpose. Symbolic computation is just one. Lisp would let you implement Mathematica with all of its functionality: windows, notebooks, language, 3d graphics, OpenGL interface and so on. This has been done for systems like Macsyma or Axiom, where Lisp is the implementation language for a system like Mathematica (including all the UI stuff).

Now, one might think further, that if for modern language to be high level (such as Mathematica, or to a lesser degree PHP, Javascript, Lisp), but if the language also needs to deal with general programming with tasks such as writing a compiler, operating system, device drivers (such as those usually done with C, C++, Java, Haskell and Lisp too), then is Lisp's internal concept of objects necessary in a language such as Lisp?

Does anyone know?

First: a notation of is always for an external representation.

What we know about languages (syntax, parsers, ...) determines how our programming languages look like. The Lisp model of a programming language is a bit odd, in that it has been shaped by the needs of working interactively with lists and other data.

Second: there is an model of data for the running Lisp system. We also have a model in our mind how the data is organized in a Lisp system. We 'see' how the internal data gets gets written to the output and we can also use functions to manipulate the data and observe effects.

So here comes the first complication: there are 'objects'. What are objects? Objects have an identity (so you can have two arrays and you can observe that they are different arrays – are they EQ or not?), a type (you can ask every object for its type), some properties (say, the array dimensions and the element-type of an array) and the data (the array contents).

So, objects have:

- * identity

Re: the necessity of Lisp's Objects?

Re: the necessity of Lisp's Objects?

- * a type
- * may have some internal information
- * data

CONS

- * you can check if to cons cells are EQ or not
- * type is cons
- * no internal information
- * CAR and CDR cells are the data

But there are other data types like that: strings, characters, hashtables, streams, pathnames, ...

Second complication: there is some data that is not really an object like above: numbers are an example.

Third complication: there are objects where you don't get access to the data. An example is a compiled function. It is a real data object, but there is no function to set or get the machine instructions of that function. You can create function objects and call them later. But you can't access the machine code. You can also not print them in a way, so that you can read it back. You would need to be able to write and read the machine instructions – which makes not that much sense for text output.

So, now we have an idea about the internal data and about internal objects. Let's print them:

a number: prints based on its types
integer: 64568405
ratio: 1/10
complex: How do we print a complex number?
Floats: how do we print a float number? That's not that easy.

symbol:
usually we want: FOO
but symbols can be arbitrary strings so we need sometimes:
|this sentence is REALLY a symbol|

CONS:

we print: (foo . bar)
what we don't print: the identity and the type.

LISTs, that's odd

there are no real lists. Lists are conses or NIL.

We could print them as conses:

(A . (B . (C . NIL)))

But for convenience we print (A B C)

PATHNAMES?

Re: the necessity of Lisp's Objects?

Re: the necessity of Lisp's Objects?

tough. Pathnames are different on various platforms.
One could give up and say they are strings.
Common Lisp has pathnames as objects with data
(host, directory, name, version, ...).
#P"/foo/bar/test.lisp"

FUNCTIONs ? How would you print the result of:
(compile nil (lambda (a) (+ a 10))) ??
It is probably compiled to some machine code.
so we do just this: #<Function 15 2009528A>

and more...

So the whole business of printing is quite complex and
we have to make all kinds of decisions how different
data gets printed. For some data we don't really
know how to print it (example: functions). For some data we
we don't want to print all detail. For some data we want
to optimize printing (lists, ...) based on some decisions.
So, Common Lisp provides a machinery for all this and
more.

Decisions the user has to make:

do you want to print the data in a way that can be read back or not?
for a string this means: double quotes around the string or not?

do you really want to print the data in length?
do you really want to print a 1000 x 1000 x 1000 array to
your text console?

do you really want to print the data in full depth?
say, you have a tree that has a depth of 100, do you want to print
all levels?

and more...

Now we come to reading.

How do we read data? READing means turning an external
representation into an internal one.

423423 -> will be read as a integer number
(A B C) -> will be read as a list. READ will create the conses and symbols (if necessary).
[this REALLY is a symbol] : we get a fresh new symbol if there isn't one.
if there is one already, we get that
"this surely is a string": we get a fresh new string object
#p"/foo/bar.lisp": we get a pathname
#<Function 15 2009528A> : what now? we can't read that.

Complication 1: we can't read everything.

Re: the necessity of Lisp's Objects?

Re: the necessity of Lisp's Objects?

We can only read what has enough information in the textual form, so that we can create data from it.

Complication 2: we don't get the original data objects back.

Complication 3: for symbols we get a unique object.

Complication 4: there is often not enough information in the printed data to get a data object back, which has all the properties of the object we printed.

All of the above (PRINTing, internal representation, READing) has to fit together in some form of machinery that is:

- * easy to use
- * extensible
- * understandable

You can see that the above is REALLY difficult and that there are lots of design choices over a period of evolution is involved. A huge hack? Maybe. But there is no easy and obvious solution. It is also not possible to see each dimension in isolation. One can't just focus on designing the PRINT functionality – one needs to know what data types there are and one also needs to know what kind of READ functionality you want to provide. The internal data representation is also not independent from the other subsystems (say I/O). Lisp's model is already higher-level than what you get from your operating system (chunks of memory and interfaces that use C data conventions (mostly)).

.