

Re: Why is LISP syntax superior?

Source: <http://coding.derkeiler.com/Archive/Lisp/comp.lang.lisp/2006-06/msg01245.html>

- *From:* "Wolfram Fenske" <int2k@xxxxxxx>
 - *Date:* 24 Jun 2006 14:29:41 -0700
-

Hrvoje Blazevic <hrvoje@xxxxxxxxxxxxxxxx> writes:

Wolfram Fenske wrote:

keke@xxxxxxx (Takehiko Abe) writes:

Ah. I guess Lisp is not for you then.

I think that either you like it or you hate it on the first sight. Do not try too hard. It's futile.

Not necessarily. At least, it wasn't for me. When I first saw lisp, I thought the designers were simply too lazy to write a real parser and that the language wasn't all that great. Much later, I stumbled upon Paul Graham's essays where he claimed over and over that lisp was the most powerful programming language ever invented, and after reading several of them, I started believing him. To see what it's all about, I bought Peter Seibel's excellent book, "Practical Common Lisp," finally got macros and now I'm something of a convert. Understanding macros is essential to appreciating the power of lisp, though. IMO, if lisp didn't have its macro system, there'd be no significant advantage over, say, Python and hardly any justification for its (lack of) syntax.

An interesting view, and quite the opposite of what makes me thick. It is precisely the lack of syntax that I admire in Lisp (Scheme actually).

Admittedly, that has a certain aesthetic beauty to it. There's hardly any syntax in the core language of CL and of Scheme, i. e. the basic evaluation rules plus the special operators. But when it comes to macros, you get additional, non-regular syntax in Lisp as well.

I think there are some cases where more syntax would be nice in Lisp. One thing I especially like in Python is it's notation for accessing

Re: Why is LISP syntax superior?

elements or subsequences of a sequence, the abstract super-type of string, list, tuple and dictionary (Python's hash table). For elements it's simply

```
elem = seq[i]
seq[i] = new_elem
```

For subsequences it's

```
sub_seq = seq[i:j]
seq[i:j] = new_sub_seq
```

I find that more convenient and easier to read than what standard CL offers, namely

```
(let ((elem (elt seq i)))
...
(setf (elt seq i) new-element))
```

```
(let ((sub-seq (subseq seq i j)))
...
(setf (subseq seq i j) new-sub-seq))
```

I guess one could write a reader macro to transform `foo[i]` into `(elt foo i)`. Hm ... Anyway, in the presence of CL's macro system, these issues become way less important for me, since I can spare myself so much stupid typing that having to write `(elt seq i)` is a small price to pay. *But:* If I were to design a new programming language without such a powerful macro system (not that I want that), I'd probably not choose S-expressions because I think other representations would be more suitable.

I did learn Python (some years ago), and did not like the extra syntax much, but most of all what made me drop Python was that at version 2.1, they still did not get the lexical scope properly (failing to create closures).

That's indeed quite disappointing.

As for really impressive syntax languages like Java, I did manage to read one 900+ Java book, but it was a really painful experience.

No doubt about that. ;—)

I had to constantly flip back not for failing to understand how exercises were supposed to be solved, but for not being able to

Re: Why is LISP syntax superior?

remember the ton of syntax that Java was throwing at me. But then again, I'm getting old, so maybe that is the problem :-)

Hehe! :-) Certainly Java isn't the best example of a language without S-expression-syntax. But there's e. g. Ruby, Python, Perl or OCaml [1].

I guess my point was, although I didn't express it very clearly in my other post, that I'm more than fine with S-expressions in Lisp because it gets me this powerful macro system. But if I didn't have that, I think I'd prefer another syntax.

Wolfram

Footnotes:

[1] OCaml actually lets you extend the parser so that you can build your own preprocessor (don't think C preprocessor here) that transforms your dialect into proper OCaml code which then gets compiled. It's not as nicely integrated as CL's macros but equally powerful.

.