Hy – A Lisp-flavored Python (readthedocs.io)

625 points by tosh on Aug 2, 2017 | hide | past | web | favorite | 215 comments

IgorPartola on Aug 2, 2017 [-]

As a heavy user of Python, and someone who grew up with the "curly braces" languages, I have a question for y'all. Is this really readable?

```
(setv result (- (/ (+ 1 3 88) 2) 8))
```

Or rather is it more readable than

```
result = ((1 + 3 + 88) / 2) - 8
```

I just... Do you just get used to this, or is it something that you have to keep struggling with? Especially given that the latter is how we do math the rest of the time?

mthomas on Aug 2, 2017 [-]

I find that both are equally (un)readable. However, for expressions that are that long, I'm inclined to write:

```
(setv result (-> (+ 1 3 88)
                 (/ 2)
                 (- 8)))
```

jamescostian on Aug 2, 2017 [-]

I am not a lisper, but that looks pretty cool. The non-lisp ways of writing that would either require adding parens like GP or writing things like "result /= 2", but that means repeating "result" or whatever variable many times.

cle on Aug 2, 2017 [-]

This is even better than in-fix notation.

btschaegg on Aug 3, 2017 [-]

Still being an absolute beginner at lispy syntax, I'll have to ask you the same question though.

While I see the awesomeness in threading macros, I'm usually left pondering a while on where the result actually goes in the next s-expression. How long did it take you to get used to that? Would you prefer someting like `as->` for readability?

klibertp on Aug 3, 2017 [-]

I thought it's a simple mnemonic: -> means inserting the form just after the beginning of the next one, while ->> inserts at the end of it.

You most certainly can have explicit variable to thread through expressions, like this:

```
(it-> (form1 it) (form2 3 it 4) ...)
```

where it means "previous expression". I think there's a Clojure library which already implements this. You can use that if you want.

serpix on Aug 3, 2017 [-]

in Clojure this is achieved with `as->`

baldfat on Aug 3, 2017 [-]

In Racket we actually have a threading macro with ~>

This:

(- (bytes-ref (string->bytes/utf-8 (symbol->string 'abc)) 1) 2)

Becomes this:

```
(~> 'abc
    symbol->string
    string->bytes/utf-8
    (bytes-ref 1)
    (- 2))
```

https://docs.racket-lang.org/threading/index.html

peatmoss on Aug 3, 2017 [-]

I believe the use of "~>" rather than "->" in Racket was set in #lang rackjure which adds in some of the Clojure syntax to Racket. Also amusing is their explanation:

> For example the threading macros are ~> and ~>> (using ~ instead of -) because Racket already uses -> for contracts. Plus as Danny Yoo pointed out to me, ~ is more "thready".

http://docs.racket-lang.org/rackjure/index.html#%28part._.Ba...

More thready indeed :-p

alphapapa on Aug 5, 2017 [-]

Emacs 25 has thread-first and thread-last macros in subr-x.el.

lliamander on Aug 3, 2017 [-]

Also, I've seen lisps that obviate the need for the outermost parentheses (I think by using significant whitespace). In that case, the above example would have as many parens as the python example (for those who are keeping count).

lliamander on Aug 3, 2017 [-]

Indeed, I like that better than either typical prefix or infix nested expressions.

What do you advise in the case the expression tree is a little bit more complex?

For example: (setv result (- (/ (+ 1 3 88) 2) (* 8 (+ 3 2)))

jlarocco on Aug 3, 2017 [-]

Because the operators all take a list of values it's sometimes convenient to format them like any other long list, with each nested sub-expression on its own line to form a tree:

```
(setv result (- (/ (+ 1 3 88)
                   2)
                (* 8
                   (+ 3 2)))
```

Another possibility is to use a (let ...) to give nested values temporary names, like:

```
(setv result (let ((first (/ (+ 1 3 88) 2))
                   (second (* 8 (+ 3 2))))
               (- first second)))
```

It helps if "first" and "second" have meaningful names like "velocity" or "force" that can describe what they are.

mthomas on Aug 3, 2017 [-]

I don't see any clever way to make that more readable. I would have to break it into two subexpressions and then take the difference. i.e.

```
(setv e0 (-> (+ 1 3 88) (/ 2)))
(setv e1 (-> (+ 3 2) (* 8)))
(setv result (- e0 e1))
```

kazinator on Aug 3, 2017 [-]

BTW doesn't *setv* support multiple pairs?

```
(setv e0 (-> (+ 1 3 88) (/ 2)))
      e1 (-> (+ 3 2) (* 8))
      result (- e0 e1))
```

like *setq* and *setf* in CL.

kazinator on Aug 2, 2017 [-]

There are advantages:

Long Lisp expressions break into multiple indented lines according to a couple of very easy rules:

```
(- (/ (+ 1 3     ;; sideways tree [-] --- [/] --- [+] -- 1 3
         8 8)    ;;                    |        |       `- 8 8
      2)         ;;                    |        ` 2
   8)            ;;                    `- 8
```

by contrast, there is any satisfactory, canonical way to break up infix into multiple lines (with excellent support from editors like Emacs and Vim). Thus, Lisp expressions can be longer while remaining understandable. (Which is important because entire function definitions, for instance, are really one big expression.)

There is no ambiguity in Lisp expressions. The parentheses are not optional grouping punctuators, but operators denoting operator application (analogous to the parentheses in f(x, y)). You never, ever have to grapple with *"what is the precedence/associativity of this?"* or *"I know the precedence/associativity of this, but did the person who wrote this intend it that way?"*

Sufficiently long expressions are hard to read in any notation. Just like in any language, Lisp programs break down complex calculations with intermediate variables:

```
(let ((bytes (+ 1 3 8 8)) ;; what is this term? Oh, bytes
      (words (/ bytes 2)) ;; why divide it by two? Ah 16 bit words
      (size (- words 8))) ;; Why subtract? Account for some header or special offset.
  ...)
```

> Do you just get used to this

What happens is that you *have* to get used to working with big, nested prefix expressions: because that's what functions are. A 200 line function in Lisp is a 200 line nested expression. You find that this it is perfectly manageable and you even like it. You get used to structuring code that way, and then that all applies to arithmetic expressions also, which are just more of the same thing. Given that you're able to cope with big functions made up of that sort of syntax, you are then not fazed by some five node arithmetic subexpression.

brudgers on Aug 2, 2017 [-]

Neither is readable because both are filled with magic numbers. With experience the Lisp becomes more readable when formatted conventionally:

```
(setv result (- (/ (+ 1 3 88)
                   2)
```

```
                8))
```

Line breaking the Python is possible but unconventional and requires additional symbols.

As expressions become more complex, languages with infix operators are faced with the choice between some form of operator precedence and strict left to right (or right to left) application. It is hard to look at:

```
6 * 89 & 57 >> 2**2 - 1
```

and see the implicit parenthesis. And maybe not having to figure out where the implicit parentheses go in complex cases is part of the attraction of Lisp syntax. Lisp code does not require going to the rule book as often.

mythrwy on Aug 3, 2017 [-]

Once you are in the parentheses line breaks in python are just fine, no additional symbols needed.

```
result = ((( 1 +

        3 +

        8) /

        2)

        * 6)
```

works.

Also works as a way to deal with long strings without the \.

```
long_string = ('hello'
    'really'
    ' really'
    ' long'
    ' line'
    )
```

brudgers on Aug 3, 2017 [-]

Line breaks in Python are generally unconventional. For what it is worth, the PEP 8 recommendation is for the operator to begin the line (Knuth style), i.e.

```
result = ((( 1
             + 3
             + 8)
             / 2)
             * 6)
```

How much that looks like Lisp is a matter of opinion I suppose.

jonahx on Aug 2, 2017 [-]

> Do you just get used to this

Short answer: yes, and in *far* less time than you'd think. The difference between the examples you cite, specifically, is trivial, and if it's a sticking point for you I'd suggest (and I'm not trying to be catty at all) that you broaden your experience with different languages. I mean, languages like J can become readable with enough experience -- stuff like the above is just a blip on the syntactical radar screen.

Optimizing syntax for readability is a worthy goal, but experience (both with the language in question and languages in general) is an almost insurmountable confound here, and ultimately your goal should be to effortlessly "see past" surface differences so you can concentrate on the deeper facilities the language provides for expression.

IgorPartola on Aug 3, 2017 [-]

Heh. I am pulling this example straight from Hy's tutorial. They are the ones claiming that this syntax is superior to Python's. I do have a hard time with parsing it in my brain.

Long ago, I internalized what I believe to be true about programming languages: they should be optimized for the humans using them, not for the machines that run the programs or the compiler authors who create them. This leads to less bugs. Maybe *your* goal is to see past syntax to understand language semantics. My goal is to get code shipped with as few bugs as possible as quickly as possible. Python and ES let me do that pretty well. Also don't meant to be catty, to each their own interests.

didibus on Aug 3, 2017 [-]

I think what was meant was that the semantics should be optimized to human. Those semantics matter more then the syntax, since you can quickly get used to syntax.

So think of it as learning to recognize and pronounce an alphabet used to form words, over understanding the meaning of sentences in a given language.

Having said that, once you get used to the lisp syntax, as I did, I find it is better suited to humans. But to believe it you need to try out structural editing.

IgorPartola on Aug 3, 2017 [-]

Hmm, that seems to imply that you can't optimize for semantics and syntax at the same time.

didibus on Aug 3, 2017 [-]

It can optimize for both, in fact that's what Lisp is all about. Take Hy, its got Python's semantics, supports all of its features and paradigms, but its syntax is simpler, more uniform, homoiconic and more powerful. This is turn allows you to extend, modify or refine Python's semantics, since the Lisp syntax will always be able to express them.

jonahx on Aug 3, 2017 [-]

> they should be optimized for the humans using them, not for the machines that run the programs or the compiler authors who create them.

Almost every high-level language espouses this goal -- even something as seemingly esoteric as J *specifically* does. Which is to say, your argument assumes that python's syntax is somehow more suited to humans, but I think you'd be hard pressed to back that up with evidence. Clearly it *feels* more natural to you, but why? It's hard to control for familiarity even in your own personal case, let alone programmers in general.

jwr on Aug 2, 2017 [-]

From a long-time Lisper (Common Lisp, and now Clojure): not, this is not comfortably readable.

That said, I very rarely write code like this. I tend to split operations over multiple lines, so that you don't spend so much time digging into and out of nesting levels. Prefix notation is not ideal for reading math. People have been writing infix math notation libraries for CL for exactly this reason.

It's not a showstopper, though. Also, I do find that sometimes having + and < to be of arbitrary arity is really convenient and makes for nicer code.

kazinator on Aug 2, 2017 [-]

> People have been writing infix math notation libraries for CL for exactly this reason.

People have *writing* these notation libraries, but then *not actually using* them.

Infix libraries for CL mostly exist to end arguments; "we have infix; Lispers choose not to use it, and you will understand why soon enough".

a-saleh on Aug 3, 2017 [-]

I actually used them quite often when i.e. dealing with math.

I just realized I don't use math that ofthen when programming :)

evilduck on Aug 2, 2017 [-]

It's just training. Buy and use a RPN calculator and lisp suddenly seems normal.

(To answer your question directly, yes it's readable. You learn to look past the syntax in lisp the same as you stop reading semi-colons and curly braces in other languages)

chx on Aug 3, 2017 [-]

Normal? It's upside down! I am a die hard RPN fan, started with a HP 32S when I was 14 years old almost three decades ago, upgraded to a HP 48G a few years old (well, kind of a 48G - 48GX hybrid, a friend of mine and I hacked it to 512K SRAM, gosh, hand soldering those little TSOP legs, brrr, we just stacked them chips and ran the data legs together, those were the days).

But this is alien and hard to read, RPN should be _reverse_ polish notation, you know? And + 1 2 3 is weird because it makes + a three argument operation, or I guess in this case, variable number of arguments. 1 2 3 + + is easy to read and understand :)

gkya on Aug 3, 2017 [-]

M-x calc!

xelxebar on Aug 3, 2017 [-]

Or if you're a fan of cli tools, check out dc!

vram22 on Aug 3, 2017 [-]

For people who prefer infix, bc was/is a more user-friendly wrapper over dc. See:

$ man bc

and

https://en.wikipedia.org/wiki/Bc_(programming_language)

I normally invoke it with:

$ bc -l

to get more digits of precision after the decimal point (see Wikipedia page).

Excerpt:

[bc first appeared in Version 6 Unix in 1975 and was written by Robert Morris and Lorinda Cherry of Bell Labs. bc was preceded by dc, an earlier arbitrary-precision calculator written by the same authors. dc could do arbitrary-precision calculations, but its reverse Polish notation (RPN) syntax was inconvenient for users, and therefore bc was written as a front-end to dc. bc was a very simple compiler (a single yacc source file with a few hundred lines), which converted the new, C-like, bc syntax into dc's postfix notation and piped the results through dc.]

Note that GNU bc is no longer a front-end to dc:

[In 1991, POSIX rigorously defined and standardized bc. Two implementations of this standard survive today: The first is the traditional Unix implementation, a front-end to dc, which survives in Unix and Plan 9 systems. The second is the free software GNU bc, first released in 1991 by Philip A. Nelson. The GNU implementation has numerous extensions beyond the POSIX standard and is no longer a front-end to dc (it is a bytecode interpreter).]

Also, you can pipe the standard output of any other (filter) command into bc, and likewise, pipe bc's output to any other filter.

unkown-unknowns on Aug 3, 2017 [-]

I use dc every now and then but I wish it'd do decimal math by default.

```
9 35 5 12 *+/p
```

Furthermore I think it's strange that the order of the operands of the division operator is divisor as top-most value on the stack and dividend as second top-most value and not the other way around.

For example, to me it'd feel more natural if

```
3 5 /p
```

was equivalent to

```
5 / 3
```

in infix notation, not

```
3 / 5
```

Because when I think of a fraction I think of it as "dividend over divisor".

Edit: Actually I Googled and it makes sense now. I've been focusing on what's on the stack but if we look at what is provided as input instead then it becomes "calculate dividend, calculate divisor, divide". Longer example which shows this:

Algebraic expression

```
(4 + 2 * 5) / (1 + 3 * 2)
```

RPN

```
4 2 5 * + 1 3 2 * + /
```

boomlinde on Aug 3, 2017 [-]

Some distributions use a ".dcrc" or some such that will be executed on startup. Dump "3k" in there to set the precision.

The order of the operands of non-commutative operators reflects popular RPN calculators and Forth. You can swap the topmost elements of the stack with "r", for example "3 5r/p"

gkya on Aug 3, 2017 [-]

Maths in lisp is like learning a language with a different alphabet, it's confusing at first, then becomes natural. I myself prefer lisp's syntax especially for maths and parenthesise heavily in other languages because in you can never know how tightly they follow mathematical association rules, and that can be a source for really annoying bugs. Generally the total absence of syntactical ambiguity in lisp is really relieving (tho lots of complexity is shifted into macro expansion, which is less intriguing for me as macroes allow easy inspection whereas core language constructs not so much).

benji-york on Aug 3, 2017 [-]

> you can never know how tightly they follow mathematical association rules

You might be interested in the way Pony (https://www.ponylang.org/) handles mathematical order of operation rules: it doesn't. Runs of like-operator are allowed, but any complex expression has to have all the operations grouped by parentheses so as to remove any ambiguity.

gnaritas on Aug 3, 2017 [-]

Smalltalk and Lisp work that way as well; operator precedence is an illogical hangover from the blackboard and chalk era; it's a needless complication that greatly simplifies both the language and understanding when removed.

klibertp on Aug 3, 2017 [-]

Yes! Now try telling this to some more mathematically oriented people. The

opposition is so violent that I stopped trying long ago :(

gnaritas on Aug 3, 2017 [-]

> mathematically oriented

i.e. the blackboard indoctrinated. It's much harder to unlearn something than to learn something, you have to show them what they think is "natural" isn't, it's just what they were taught and it's based on laziness not intelligence and implicit hidden rules are bad and lead to ambiguity. Operator precedence is simply stupid. In Lisp and Smalltalk, operators are just ordinary functions and the lack of precedence makes everything simpler across the board.

dragonwriter on Aug 2, 2017 [-]

Prefix notation is something you get used to; it's not as easy as infix for operations that use the same infix operators with the same meaning and priority in the domain (e.g., math) but real programming languages often use operators that aren't standard math operators and have precedence and/or associativity quirks with the operators they use which increase overhead. Prefix notation is basically everything-is-a-function, which I think is ultimately less overhead *overall*, though still more for *simple* math, than infix.

Indentation can help, too.

6581 on Aug 2, 2017 [-]

I have been using RPN calculators for all my life (except when I was forced to use infix calculators in school). Your first example is at least as readable to me as the second one. Just read it from inside to outside instead of from left to right.

dunham on Aug 3, 2017 [-]

In school, RPN occasionally caused trouble for me as I could work through an entire problem (leaving values on the stack) without writing down any intermediate steps.

rthomas6 on Aug 4, 2017 [-]

As a Python user who read about 60% of SICP, yes it really is readable.

The trick is to realize that in lisps open-parenthesis denotes something more than a grouping. (means "function call", and a change in scope.

So if it really was written in Python, it would look something like this:

```
setv(result, sub(div(add(1, 2, 88), 2), 8))
```

Really, this is clearer what's happening under the hood than infix notation. + is a function, and so is =. Why is it called the way that it is, other than convention? It's just syntactic sugar. Doing it with prefix notation makes every function have the same syntax, which makes nesting functions easier to look at and reason about.

breck on Aug 2, 2017 [-]

I'm a Lisper (and Pythoner) and strongly agree, the first one is not readable. I would say it's gross. It boggles my mind that Lispers haven't taken that problem seriously. I have taken it very seriously and have been working on it for years.

I just started publishing my results.

http://breckyunits.com/the-flaw-in-lisp.html

What I've discovered is that the parentheses are completely unnecessary in Lisp, and in fact cause much harm by making Lisp harder to read and maintain in the long run. Drop the parentheses in Lisp by switching to whitespace/geometry for syntactic structure and you suddenly have the dream language.

Your example above would be something like:

```
setv result
 -
```

```
/
 + 1  3  88
 2
8
```

You might think that looks a little strange too at first, but the neat thing about that code is you can write it on a piece of graph paper, draw lines around each word, connect the indented nodes, and you'd have a tree visualization of your syntax code without doing a single transformation of the source.

kod on Aug 2, 2017 [-]

Yeah, whitespace sensitive languages are a great idea...

Until you permanently lose user data because of a pull request that contained whitespace changes.

Yes, I've seen this happen in production, yes it could have been prevented in other ways... but why invite disasters like that upon yourself?

simonh on Aug 3, 2017 [-]

Have you really never seen a bug in curly brace code due to a misplaced curly bracket hidden by incorrect indenting? Or code that was just really difficult to understand and modify accurately because it was indented inconsistently with the braces? And yes there are tools that can help with that, but...

I worked on Quartz for a few tears, that's over 10 million lines of Python code and many thousands of commits per day. If concerns like that really were anything more than nitpicking, projects like that would know about it.

kazinator on Aug 3, 2017 [-]

Discrepancy between actual code structure and indentation is susceptible to diagnosis by machine. That's the beauty of having a bit of redundancy in the representation.

Recent GCC now has a warning for indentation that doesn't match code structure. (Had to recently patch a breakage in GNU Binutils in an embedded distro due to this being a -Werror).

kod on Aug 3, 2017 [-]

The point is with lisp you can automatically and unambiguously parse and format code regardless of whitespace. Giving that up in favor of whitespace sensitive languages has real world drawbacks, no matter how many "no true python programmer" arguments you make.

JadeNB on Aug 3, 2017 [-]

> I worked on Quartz for a few tears

This lovely typo deserves to be noticed.

mulmen on Aug 2, 2017 [-]

Is that a problem with whitespace or your test/qa/release process? A curly brace language isn't any less likely to lose user data to a bug just because it has curly braces.

kod on Aug 3, 2017 [-]

It is less likely to cause problems, because a removed curly brace won't parse correctly, and won't be hidden when someone excludes whitesoace from a diff.

JadeNB on Aug 3, 2017 [-]

> won't be hidden when someone excludes whitesoace from a diff.

Surely it is good advice not to exclude whitespace from a diff when using a whitespace-sensitive language?

pharrington on Aug 3, 2017 [-]

Your -if- clause sans curly braces definitely still parses.

jjnoakes on Aug 3, 2017 [-]

Depends on the language.

lgas on Aug 3, 2017 [-]

Technically true, but I mean c'mon... Java, JavaScript, C, C++, C#, PHP... You have to get down to into the Go/Swift neighborhood before you'll find mandatory braces.

breck on Aug 2, 2017 [-]

This is a great concern, and glad you brought it up.

In ETNs (the new family of programming languages I'm talking about), there are no "whitespace changes", in the sense that all changes to the code affect actual nodes in the program.

This is a really important property that embeds diffs with a whole lot more meaning. You can not only see the number of lines changed, but also the number of nodes changed. If you expected a change to only change content and not change any nodes, and you saw in a negative number in the nodes changed field, you would know immediately that something had gone wrong.

This is one of the many new beneficial properties unique to these languages.

kod on Aug 3, 2017 [-]

Unless your language is somehow changing the behavior of existing tools like diff -w or github ?w=1, you're just making semantic distinctions.

breck on Aug 3, 2017 [-]

Correct. We have new diff tools specifically for ETNs.

gnaritas on Aug 3, 2017 [-]

No, that's not changing the behavior of the existing tools everyone uses and aren't going to stop using. You can't say correct when you're not actually agreeing with his point.

breck on Aug 3, 2017 [-]

Perhaps I misunderstood the parent comment.

It changes the behavior in the sense that there is no w=1 option. All whitespace is significant. The idea of ignoring whitespace would be like the idea of ignoring angle brackets in html. Whitespace is an essential part of TN and ETNs and is never ignored.

Diff tools then can benefit by showing not only aggregate line diffs but aggregate node and word level diffs without knowing anything about the meaning of a program.

olavk on Aug 3, 2017 [-]

Well, if you randomly delete parentheses in Lisp or curly braces in C you will also get problems. So...don't delete information at random.

anaphor on Aug 2, 2017 [-]

The problem is that it requires all of your functions to have a fixed arity I think, or at least a lower bound on the arity, or you use newlines as part of the syntax. Check out this

https://shriram.github.io/p4p/

breck on Aug 2, 2017 [-]

Very interesting link, thanks for sharing. I'm always eager to see developments in the Racket world.

> The problem is that it requires all of your functions to have a fixed arity I think, or at least a lower bound on the arity, or you use newlines as part of the syntax.

This was my initial thought as well.

However, in my latest batch of ETNs (the name I give to these types of lisps), I've found that a simple convention of putting any variadic argument last seems to work well.

An example may help. Here would be a language blueprint that defines 2 node types:

```
=+2
 description Adds two numbers
 pattern resultIdentifier number number
=+
 description A variadic adder of unknown arity
 pattern resultIdentifier number+
```

Here's a sample program in the language we partially defined above:

```
=+2 sum 2 5
print sum
# prints 7

=+ sum 1 2 4
print sum
# prints 7
```

flavio81 on Aug 3, 2017 [-]

>I'm a Lisper (and Pythoner) (...) It boggles my mind that Lispers haven't taken that problem seriously. I have taken it very seriously and have been working on it for years. I just started publishing my results. http://breckyunits.com/the-flaw-in-lisp.html

Honestly, there is no flaw in the parentheses. You just need to write more Lisp.

The parentheses are one of the *best* things in Lisp.

breck on Aug 3, 2017 [-]

Disagree. The parens are a missed opportunity. If you use geometry instead, you gain extraordinary new powers of analyzing and editing large programs in 2 and 3 dimensions. Not possible with the parens. The benefits still are not obvious but we're working on publishing more data and tools

flavio81 on Aug 4, 2017 [-]

Well, then have fun programming in Befunge, exactly the programming language that will fullfill your bidimensional desires.

breck on Aug 5, 2017 [-]

Sadly, I've worked with some code bases before that may have been more fun to work on if they were in Befunge.

blain_the_train on Aug 3, 2017 [-]

Take a look an Parinfer https://shaunlebron.github.io/parinfer/. Which essential makes clojure a whitespace language with indent mode.

breck on Aug 3, 2017 [-]

Super interesting.

I wonder if the creator has considered just not having parens at all?

undershirt on Aug 3, 2017 [-]

I have! I think and so have others:

- http://nonelang.readthedocs.io/en/latest/dataformat.html#nak... - http://readable.sourceforge.net/ - https://github.com/boxed/indent-clj

breck on Aug 3, 2017 [-]

Interesting, I should have been clearer when I said not just no parens, but also just whitespace instead of any syntax characters. The none language uses both () and "". Sweet expressions use {}. And indent-clj uses () and [].

blain_the_train on Aug 3, 2017 [-]

I have had the same thought. I'm guessing, given how long lisp has been around. That the ideas is well research.

dragonwriter on Aug 3, 2017 [-]

> Drop the parentheses in Lisp by switching to whitespace/geometry for syntactic structure and you suddenly have the dream language.

This has been "discovered" many times, but somehow this dream language never approaches even Lisp in popularity. (Though a loosely similar transformation of it's basic ML-based syntax is used in Haskell to reduce parens, but that's not a Lisp base.)

breck on Aug 3, 2017 [-]

> This has been "discovered" many times,

So far I'm only aware of one previous discovery--https://srfi.schemers.org/srfi-49/srfi-49.html--, which nsajko pointed me to a couple weeks ago, (https://news.ycombinator.com/item?id=14606595). Are you aware of other identical or near identical implementations? The tiniest details matter here. I'd be very interested in more.

> but somehow this dream language never approaches even Lisp in popularity.

Agreed! That's what I'm hoping to help change. It's going to be a long slog, but I'm very confident the math works out and it's going to happen. The tooling needs to be built to take advantage of the properties of these languages before they will catch on. But once that happens, I think we're in for a whole new world in programming.

kbp on Aug 3, 2017 [-]

Logo lets you leave out parentheses most of the time, except to use rest arguments or force grouping. For example:

```
? print list sum 0 1 2
1 2
```

That expression parses as (print (list (sum 0 1) 2)) because both list and sum normally take 2 arguments. However, using the parenthesised function call syntax, you can pass them however many you like:

```
? (print (list 1 2 3 (sum 1 1 1 1)))
1 2 3 4
```

Or you can of course mix and match (probably to better effect than this example):

```
? print (list 1 2 sum 1 2 4)
1 2 3 4
```

Logo also lets you use infix arithmetic operators with normal precedence:

```
? print 1 + 2 * 3 + 4
11
```

breck on Aug 3, 2017 [-]

I love Logo! Advanced Logo by Michael Friendly is one of my programming books.

Logo came very close to TN and ETNs but adding the parens and other visible syntax I believe was a design mistake. It's not obvious, but when working with large programs in 2 and 3 dimensions (only possible with parens free lisp, or ETNs as we call them), you gain tremendous productivity benefits

dTal on Aug 4, 2017 [-]

>Are you aware of other identical or near identical implementations?:

srfi-110, srfi-119.

breck on Aug 5, 2017 [-]

Thanks!

Some thoughts on 110 and 119 in my comment below.

emmelaich on Aug 3, 2017 [-]

e.g. http://chrisdone.com/z/

Fascinating but I don't think it has many users!

A quote from that page...

```
(Note: there are no parentheses in Z. Zero.)

Which, if you think about it, is the natural
grouping for the definition of the name
argument syntax I gave above.

To pass additional arguments to a function,
the arguments are put on the next line and
indented to the column of the first argument:
```

breck on Aug 3, 2017 [-]

I love Z but think there are a few details that are off.

1. A single space for indenting nodes works much better. 2. strings don't need quotes.

amelius on Aug 3, 2017 [-]

I'm using this same approach in a side project. The idea is very old. See e.g. [1] and the references therein.

[1] https://srfi.schemers.org/srfi-119/srfi-119.html

breck on Aug 3, 2017 [-]

Very cool! Is wisp yours? I learned of SRFI 49 just over a month ago. I reached out to Egil and hope will be able to chat with him this summer.

As 110 and 119 also note, Egil was on to something. Although I came to it from a different angle, I think he had it about 90% correct.

My contribution is identifying the split between TN and ETNs, and then getting the details perfect with TN. Getting the details perfect meant showing how to craft it so that every single input was a valid TN program. My motivation was, if you can write it/build it in real physical life (3D world), or in other words, if nature allows it, then why shouldn't your

compiler? I think 49 missed a few tiny details (like " It is an error if neither one of the leading space/tab seqquences are a prefix of the other, nor are they "...in my TN, there are no errors, and extra spaces are translated into separators for empty words).

gnaritas on Aug 3, 2017 [-]

> by switching to whitespace/geometry for syntactic structure and you suddenly have the dream language.

Nope, you've just made it awful, white-space with syntactic meaning is just god-awful. It's the worse thing about Python, there's a reason it's not a more popular language feature.

breck on Aug 3, 2017 [-]

I actually agree.

Unless done perfectly, whitespace with syntactic meaning languages are quite annoying.

If I had a nickel for every time I had a tab vs space bug in Python when I was first starting to code.

kirbyfan64sos on Aug 4, 2017 [-]

Well, now Python 3 doesn't allow tabs anyway. ;)

breck on Aug 5, 2017 [-]

I got it excited when I read your comment, but turns out it still allows tabs, just not mixing. :(

pvdebbe on Aug 3, 2017 [-]

The thing is that usually basic algebra like this occurs somewhat seldomly in code. The rest of the function calls are readable, this is the (only) outlier to readability.

zeveb on Aug 2, 2017 [-]

I'd say that they're about equally (un)readable. What do you think of:

```
(setf result (make-instance 'foo :bar (baz quux)))
```

vice:

```
result = new Foo(bar=baz(quux))
```

The first is simple to write a parser for, but the second is … evil.

dustingetz on Aug 2, 2017 [-]

In clojure you write it like this, so the parens indicate scope. No need to mutate an environment here. dunno about hy

```
(let [result (- (/ (+ 1 3 88) 2) 8)]
  ; result defined inside the expression only
  )
```

eggy on Aug 3, 2017 [-]

I am a polyglot, so no, I don't find it unreadable. I would probably structure it differently.

I always wonder why Lisp never ceases to get singled out for parentheses when each language uses parentheses, braces, semicolons, colons, pipes, etc...

I program in C, Forth\Factor, Lisp/Scheme, Python, J, APL, F#, C#, Asm, Basic, Clojure and others. Each has their syntax and differences, and it just takes getting used to them.

I like Hy, but as the documentation says it is not Lisp or Clojure, but Homoiconic Python with that guiding and constraining its implementation. A lot of fun, really. I only get as much pleasure from J for

play.

kazinator on Aug 4, 2017 [-]

> I always wonder why Lisp never ceases to get singled out for parentheses

POSIX shell:

```
FOO=$(basename $(dirname $(cat file)))
```

Make:

```
OBJS=$(foreach x,$(WHATEVER),\
        $(addprefix $(call BLAH,arg),....))
```

swiley on Aug 2, 2017 [-]

I'd imagine it's like FORTH: If you read enough of it (it only takes a few days of reading) it becomes clear.

throwaway135634 on Aug 3, 2017 [-]

In my experience, the hardest part about reading Forth is recognizing when a word *doesn't* use RPN because of metaprogramming

flavio81 on Aug 3, 2017 [-]

> Do you just get used to this, or is it something that you have to keep struggling with?

You get easily used to parenthesis and you start loving them, since the auto-indenting (on any good Lisp editor) can make you have very readable code. In fact, Common Lisp, having so much versatility in control flow operators (and, in general, a ton of handy helper functions), lets you write really readable code, and often that is the case. So highly readable, that the downside is that the common Common lisper (pun intended) feels no need to add comments in the code...

As for your formula

(setv result (- (/ (+ 1 3 88) 2) 8))

It can be written in a more readable way. As user "brudgers" wrote above:

```
(- (/ (+ 1 3 88)
      2)
   8)
```

... and it gets more evident.

As for "result", usually in Lisp you don't need to have such a variable since the last expression returned by a function automatically becomes the return value of the function.

didibus on Aug 3, 2017 [-]

I got used to it, now I almost have the reverse problem.

Having said that, sometime for math, infix operators for complex equations are easier, and its easy to write a macro in Lisp to let you do that too:

(setv result (infix ((1 + 3 + 88) / 2) - 8))

ohyes on Aug 3, 2017 [-]

I was never any good at math in the first place, order of operations always screwed me up, still does. Order of operations actually makes no sense at all (if you really think about it). So yeah, lisp is easier for me.

ohyes on Aug 3, 2017 [-]

Compare to:

1 + 3 + 88 / 2 - 2

Or (1 + 3 + 88) / 2 - 2

Or (1 + 3 + 88) / (2 - 2)

If it comes down to it, I'm going to fully parenthesize the stupid thing anything else is difficult to parse. If I'm using all the parens, prefix is fine.

endgame on Aug 3, 2017 [-]

I've only written a small amount of elisp but I've found that while you don't actually do all that much arithmetic manipulation, the ease of manipulating S-Expressions outweighs the price of entry.

a-saleh on Aug 3, 2017 [-]

Well, when using clojure and needed to deal with lot of math I usually used some sort of infix macro, i.e. [1].

Otherwise, I wasn't really bothered by the brackets. And most of the time I would be using some sort of threading macro anyway [2], such as -> mthomas suggested in one of the responses.

[1] https://github.com/rm-hull/infix [2] https://clojure.org/guides/threading_macros

jrs95 on Aug 2, 2017 [-]

I find it difficult to read if I just try and read it left to right, but if I simply go directly to the innermost operation its not really much more difficult for me. And I'm not a lisp user, either.

IgorPartola on Aug 3, 2017 [-]

Since I can't edit, I'll reply to my own comment. This is not *my* example, but one I pulled straight from Hy's tutorial at https://hy.readthedocs.io/en/stable/tutorial.html#basic-intr.... I don't think I could have dreamt up code like this.

cutler on Aug 2, 2017 [-]

It's not merely a matter of syntax or taste. Lisp's code-as-data and advanced macros are made possible by its s-expressions.

deathanatos on Aug 2, 2017 [-]

This is the main reason Lisp has just never grown on me. I'm certain — and other responses demonstrate — that the readability can certainly be improved here. Yes, a programming language needs nothing more than parens. But dedicating a tastefully chosen set of sigils seems to make a lot of difference, in my opinion.

kazinator on Aug 2, 2017 [-]

In the dialect called TXR Lisp, I have introduced precisely that: some tastefully chosen syntactic sugars which do not disturb the structure of the surrounding Lisp. I believe I made a wise choice by not targetting arithmetic. My syntactic inventions target areas like working with arrays, string processing, OOP and higher order functions.

Small taste:

```
1> (let ((x "abc") (y "def"))
     (swap [x 0..2] [y 2..:])
     (list x y))
("fc" "deab")

2> (mapcar (ret `@1-@2`) "abc" "def")
("a-d" "b-e" "c-f")

3> '#"lorem ipsum dolor sit"
("lorem" "ipsum" "dolor" "sit")

4> (let ((a "dolor")) ^#`lorem ipsum @,a sit`)
#`lorem ipsum dolor sit`
```

```
5> (eval *4)
("lorem" "ipsum" "dolor" "sit")

7> (defstruct (point x y) nil (x 0) (y 0))
#<struct-type point>

8> (new point)
#S(point x 0 y 0)

9> (new (point 1 2))
#S(point x 1 y 2)

10> (new point x 2 y 3)
#S(point x 2 y 3)

11> (let ((a (new (point 10 20))))
      ^(,a.x ,a.y))
(10 20)

12> (let ((a (new (point 10 20))))
      `<@{a.x}, @{a.y}>`)
"<10, 20>"

13> (defmeth point distance (p) (sqrt (+ (* p.x p.x) (* p.y p.y))))
(meth point distance)

14> (new (point 3 4)).(distance)
5.0

15> (defmeth point distance-to (p q)
      (let ((r (new (point (- q.x p.x)
                           (- q.y p.y)))))
        r.(distance)))
(meth point distance-to)

16> (new (point 3 4)).(distance-to (new (point 20 30)))
31.0644491340181

19> (let ((pts (build (dotimes (i 10) (add (new (point (rand 10)
                                                       (rand 10))))))))
      pts)
(#S(point x 5 y 6) #S(point x 1 y 1) #S(point x 1 y 3) #S(point x 5 y 7)
 #S(point x 2 y 8) #S(point x 7 y 3) #S(point x 1 y 1) #S(point x 2 y 4)
 #S(point x 5 y 6) #S(point x 5 y 3))

20> (mapcar .(distance) *19)
(7.81024967590665 1.4142135623731 3.16227766016838 8.60232526704263
 8.24621125123532 7.61577310586391 1.4142135623731 4.47213595499958
 7.81024967590665 5.8309518948453)

21> (mapcar (ret @1.(distance-to @2)) *19 (cdr *19))
(6.40312423743285 2.0 5.65685424949238 3.16227766016838 7.07106781186548
 6.32455532033676 3.16227766016838 3.60555127546399 3.0)
```

kbp on Aug 2, 2017 [-]

I understand this point of view if most of your programs are just doing arithmetic, but once you start using functions and procedures the differences are pretty minimal.

ifelsehow on Aug 2, 2017 [-]

As @jwr says, there are different paradigms in Lisp, such that you would never really write code like this.

jlarocco on Aug 3, 2017 [-]

It takes some getting used to, but eventually the Lisp style is just as readable as the non-Lisp style.

For actual coding (in an editor vs reading in a browser) it's even easier with things like matching parenthesis highlight, jump to matching brace, and rainbow parenthesis mode.

cannonpr on Aug 2, 2017 [-]

I used to be mostly a Python guy, fell in love with Clojures syntax, you get used to it surprisingly fast, and it's caused me less headaches than 'interesting' pythonic syntax. That having been said, I still love Python too.

erikpukinskis on Aug 2, 2017 [-]

I would never use a variable name like result, it's meaningless. And the inner values should be labeled too, with their own variable names. Any compiler worth its salt will optimize the names away.

dmoney on Aug 3, 2017 [-]

As a Python-lover who previously played with Common Lisp and Clojure, lispy syntax took less getting used to than having no end tags, but I'm still not used to prefix math.

klibertp on Aug 3, 2017 [-]

I have a blog post about this: http://klibert.eu/posts/tools_for_lisp_syntax.html (with pictures: http://klibert.eu/statics/images/lisp-editor-features1.png).

Basically, in my opinion, what makes a difference in readability is the tooling a certain kind of syntax enables. As you can see on my "Languages" page I have been trying out new languages for quite a while now, and so I've seen all levels of syntactic helpers. The syntax itself is actually a secondary concern: it's really not that important if your editor can easily generate it, manipulate it, and display while hiding parts of it for you.

However, how easy it is to write such tooling for a language depends on the kind of syntax it uses. Languages with more complex grammars are generally harder to support properly in the editor. Of course, how good the tools are, depends also on a level of effort invested in making them so - so relatively complex, but popular, languages may still have better editor support.

Anyway, back on topic of Lisp: yes, you get used to the prefix notation to the point that - after a long while, mind you - you're able to open a Lisp file in Notepad and still easily read it. No one does it, though, unless forced for some reason - things like:

- blink the matching paren, display parens as rainbow colored
- jump to matching paren
- jump to inside next expression
- reliably select current block of code
- fix indentation automatically when copy&pasting code
- split ((..|..) -> (..)|(..)), join, transpose, extend the current expression
- convolute, which means making the nested (inner) expression the outside one

all of this makes Lisp readable, and probably more readable than some other languages with much more complex grammars. Lisp is not the only language with such a "feature" - recently I found Idris to be very good in this regard, both Erlang and (even more so) Elixir are similar, some (popular) brace languages too.

Then there's a thing about macros and trivial AST manipulation in Lisp, arguably enabled by the syntax being as it is. You can have pattern-based macros in any language, actually (see Nim, or Dylan), but the raw AST being identical to the code makes certain kinds of imperative macros much easier to write. Again, take a look at Nim: is has imperative macro facility, but for every piece of syntax you want to introduce you need to call a special constructor.

platform on Aug 3, 2017 [-]

prefix notation mimics of how we call functions

```
setv result  (my_function  arg1, arg2)
```

to me this is as readeable as

```
set result= my_function( arg1, arg2)
```

I realize that specifically arithmetic expressions in prefix notation look foreign, however I think most of the code are function calls -- and that's very normal.

If I would to think of an example where LISP notation is un natural -- it would be the if-statements.

asimjalis on Aug 3, 2017 [-]

Here is how I would write that expression in Hy.

```
(-> 1 (+ 3) (+ 88) (/ 2) (- 8))
```

Here is how I would write the assignment, using my definition of def2.

```
(defmacro def2 [value variable] `(def ~variable ~value))
(-> 1 (+ 3) (+ 88) (/ 2) (- 8) (def2 result))
```

cgag on Aug 2, 2017 [-]

I find it pretty readable.

such_a_casual on Aug 3, 2017 [-]

Prefix notation is a problem because nothing beats the familiarity of:

```
x = 10
```

But prefix notation is superior. You don't have to invent an order of operations out of thin air. All your functions take several arguments, not just 2. So + is now sum, - is difference, / quotient, > greater than order, etc. They aren't special. they're just like everything else.

It'll never be as familiar, but it's smarter and makes writing code easier. For example

```
"hello " + name + ", welcome to the " + place +"."
+ "hello " name ", welcome to the " place "."
```

The reality is you want functions that work on many things, not just 2, even when you're doing math.

```
1 + height + x + floor-height
+ 1 height x floor-height
```

infix notation does nothing to make code clearer

kazinator on Aug 3, 2017 [-]

Anyway, you want string interpolation for the former example anyway:

```
1> (let ((name "Alice") (place "The Palace"))
     `Hello @name, welcome to @place!`)
"Hello Alice, welcome to The Palace!"
```

string interpolation is compatible with Lisp designs; it is self-contained and so doesn't "disturb" the surrounding syntax with issues of precedence and associativity. The above example is from TXR Lisp which has this built in. For Common Lisp, there is the *cl-interpol* and other packages, and other dialects have their own solutions.

such_a_casual on Aug 3, 2017 [-]

And it's pretty trivial to roll your own.

torrent-of-ions on Aug 3, 2017 [-]

Yes, it is. I can't say anything about being "more readable" but English is presumably "more readable" than isiXhosa for you. Reading is a skill that has to be learnt and practised. You were not born with the ability to read English or infix notation, you just got used to them.

taneem on Aug 2, 2017 [-]

The online demo is amusing/creative: https://try-hy.appspot.com/

alschwalm on Aug 2, 2017 [-]

Hy is a great project. One unfortunate thing, though, is that no one has been able to successfully implement `let`, which makes writing idiomatic lisp almost impossible. I'm curious if anyone can actually prove it can't be done or is just very difficult (or too slow to be useful).

aidenn0 on Aug 2, 2017 [-]

Is there something about Hy's lambda that makes implementing let with it not possible?

For those unaware, the transformation is roughly:

```
(let ((foo bar) ...) Y) -> ((lambda (foo ...) Y) bar ...)
```

[edit]

Reading around it seems that it breaks several pythonic control-transfer functions that treat functions specially (e.g. yield from inside a let would be surprising).

Without knowing much about Hy's semantics, it should still be doable by rewriting variable names. That's likely not doable in a robust manner via a macro though, so would have to be part of Hy's transformation from sexpr to ASTs.

shuzchen on Aug 2, 2017 [-]

Hy actually used to have a let (but a broken one) and it was removed in a recent version. For context on this see https://github.com/hylang/hy/issues/1056

nerdponx on Aug 2, 2017 [-]

What is it about `let` that makes it so hard to implement?

alschwalm on Aug 2, 2017 [-]

There is a description of the problem here: https://github.com/hylang/hy/issues/1056

I think it really comes down to python having limited scoping capabilities (I think you can only really make new scopes with function declarations), this can maybe be hacked around, but at too high a performance cost to be usable.

tuturto on Aug 3, 2017 [-]

I actually wrote a short blog post about this just recently: https://engineersjourney.wordpress.com/2017/08/01/history-of...

In short, let kept running into trouble with yield, exceptions, breaking out of loops and such. It would have been really tricky (probably close to impossible) to get everything working correctly.

bshanks on Aug 3, 2017 [-]

In case i ever find myself designing a programming language, what facilities would Python have to support in order to allow 'let' to work? It seems to me that in theory Hy could solve the problem with which variables are to be marked 'nonlocal' by doing a lot of additional code analysis (is this correct?).

I gather that the main remaining problem is that break, continue, yield, async, await work differently if you covertly introduce a new function scope. Would it be sufficient if scopes could have explicit labels and break, continue, yield took an argument that said which label to break/continue/yield out of? Would that solve the problem with async, await too?

If Hy did extensive code analysis to determine where to introduce 'nonlocal', and if Python's break/continue/yield took this extra argument, would implementation of 'let' by Hy then become possible or are there still additional impediments that i am missing?

bshanks on Aug 6, 2017 [-]

(tuturto responded to the parent of this comment at https://engineersjourney.wordpress.com/2017/08/01/history-of...)

thaumasiotes on Aug 2, 2017 [-]

I'm curious too. Conceptually, `let` defines a function of n variables and then invokes it on a list of n values:

```
(let ((a 3)
      (b 5)
      (c -1))
  (* a (+ b c)))

def «gensym»(a,b,c):
    return (a * (b + c))
«the-gensym»(3,5,-1)
```

It seems workable purely in terms of the code transformation. But, I haven't tried.

Tenobrus on Aug 2, 2017 [-]

I think the problem is, then constructs such as yield (the behavior of which depends on the current function scope), would no longer work correctly.

nerdponx on Aug 2, 2017 [-]

Couldn't `let` be a compile-time expansion using `gensym` or some other symbol mangling? Or is that what the contrib macro does?

rcarmo on Aug 2, 2017 [-]

This. See my other comment for a sizable Hy example and more info, but the lack of "let" mostly killed it for me.

nnq on Aug 2, 2017 [-]

> but the lack of "let" mostly killed it for me

Can you give an example of code that is improved by *let* significantly? ...and can't be rewritten in a more readable way without it?

hk__2 on Aug 2, 2017 [-]

> Can you give an example of code that is improved by let significantly

Any piece of code that uses a variable.

nnq on Aug 2, 2017 [-]

I'd be more inclined to day that *let* mostly encourages you to write zillion-line functions (instead of lots of small ones), simply because with it you "mentally can" since you can see the scope of variables better.

But.. do you really prefer to write huge functions instead of smaller ones?

Also: I think Python's default args being evaluated at function definition time kills the need for the usage of let when defining closures inside loops (you can just pas the value you want to capture in a closure as the default value of a param) or other things like that.

hk__2 on Aug 2, 2017 [-]

> I'd be more inclined to [s]ay that *let* mostly encourages you to write zillion-line functions (instead of lots of small ones), simply because with it you "mentally can" since you can see the scope of variables better.

That’s true. I revise my comment as "Any piece of code that uses a variable[to hold something you use multiple times and don’t want to recompute]."; for

example creating an API client then using it twice.

aidenn0 on Aug 3, 2017 [-]

Any macros that expands to code that will reuse a value. You can't do common, and valuable, constructs like *once-only* without *let*.

nnq on Aug 3, 2017 [-]

Hmm. Can't argue against this. I agree, this is a legitimate case.

Probably I'm a bit too stuck on "the python way" on this clash-of-cultures to see such cases that are clearly part of "the lisp way"...

flavio81 on Aug 3, 2017 [-]

> Can you give an example of code that is improved by let significantly?

"Let" is one of the most important special operators in Lisp. It allows you to create lexical variables within an s-expression.

Roughly equivalent to creating local variables within a block (in other languages).

So i'd claim it is *essential*.

such_a_casual on Aug 3, 2017 [-]

```
(defun udpate ()
 (let ((pos (find-char-pos)))
   (move-char pos))
 (print (stats))
 (rotate enemies))

(defun udpate ()
 (set 'pos (find-char-pos))
 (move-char pos)
 (print (stats))
 (rotate enemies))
```

I now have to keep this 'pos symbol in my head for the rest of the function (in another language, forever in this case), whereas in the first example I can just forget about the variable after the next line. It doesn't exist anymore. It makes code clearer and much easier to refactor since I can immediately see everywhere that the variable is used.

nnq on Aug 3, 2017 [-]

I get it, despite having a small issue with "forever" being defined as "just 2 more loc", but I just:

(1) feel more drawn to "Flat is better than nested." from "Zen of Python" to dislike the extra scoping level despite the conceptual clarity it provides (because I'm *sure* some "idiot" will abuse it and I'll end up reading a 10 nested let-s monstrosity pretty soon)

(2) like to write small functions, so forever will always be "~10 loc max" for me

...so imho if you're used to "thinking in classic Lisps" you'll miss *let*, otherwise you won't. What I'd miss for example would be some Haskell-like *where* (https://wiki.haskell.org/Let_vs._Where#Advantages_of_where) but I'd wake quickly out of it since it would make no sense to someone not already "stuck in Haskell-like thinking mode".

jazzdev on Aug 5, 2017 [-]

```
(defun udpate ()
 (ag-if (find-char-pos)
   (move-char it))
```

```
(print (stats))
(rotate enemies))
```

Doesn't Hy's anaphoric macros solve this? And one could even argue it's more readable.

such_a_casual on Aug 7, 2017 [-]

I don't know what that is (tried googling it), but it doesn't look like it's doing the same thing as let.

smaili on Aug 2, 2017 [-]

Excerpt:

> Hy is a wonderful dialect of Lisp that's embedded in Python.

> Since Hy transforms its Lisp code into the Python Abstract Syntax Tree, you have the whole beautiful world of Python at your fingertips, in Lisp form!

macdice on Aug 3, 2017 [-]

Interesting that it uses defun, like Common Lisp, the main Lisp-2 dialect, and yet it's clearly a Lisp-1, like Scheme. After (defun f (n) (+ n 1)), evaluating f shows that it's a function (just as in Python, which is a Lisp-1). My first thought, for what it's worth, is that if the namespace semantics is like Scheme it'd be better to follow Scheme idioms rather than Common Lisp, so code can be easily ported. On the other hand, this is only my first minute using Hy so what do I know. Looks really neat!

kazinator on Aug 3, 2017 [-]

EuLisp (pretty much defunct) is also a Lisp-1 dialect with *defun*.

etiam on Aug 2, 2017 [-]

There's even a couple of kernels for IPython/Jupyter. The best I've seen so far is https://github.com/Calysto/calysto_hy

I'd like to move to Hy for much more of what I currently do in Python, but so far I've been too lazy to find/create good editor support for it. For actual projects of some size the loss of context help, documentation, various completions, etc seems like too high a price to pay.

zitterbewegung on Aug 2, 2017 [-]

I think the highest cost is really collaboration with others. If you ever want help with your project or if people even want to use it you are dealing with only Hy users and disregarding regular Python users. I do like the concept of Hy though.

etiam on Aug 2, 2017 [-]

It's restrictive that way, no doubt.

On the other hand, I suspect that as subsets go, Hy programmers would be a good one.

And I'm not sure the impact on getting people to use it would necessarily be *that* bad. It's just an extra import hy away to use modules written in Hy just as usual from Python, after all.

thro1237 on Aug 2, 2017 [-]

Also, in terms of useful debug/error messages. That was the biggest drawback of clojure.

winter_blue on Aug 2, 2017 [-]

Since Python doesn't support multi-line lambdas, how do they support the analog in LISP? It's almost necessary for any dialect of LISP to support lambdas (i.e. functions) that contain a LISP 'do' [1] which lets you group statements. Do they chain together expressions/statements with continuations or something?

[1] http://www.lispworks.com/documentation/lw60/CLHS/Body/m_do_d...

joejev on Aug 2, 2017 [-]

Python lambdas are just syntactic sugar for creating a function object. The ast can just build a named function and use it in the needed expression.

> kazinator on Aug 2, 2017 [-]
>
> However, that approach seems rather backwards compared to having an anonymous function primitive which is used as the basis for the lambda one-line-only sugar, as well as named functions (which simply establish a binding between a name and the anonymous thing).
>
> > aoeusnth1 on Aug 2, 2017 [-]
> >
> > Why should the programmmer care about whether the underlying function is named or not?
> >
> > > jwdunne on Aug 2, 2017 [-]
> > >
> > > I imagine some kind of analogue to gensym would work there. The name can be throw away and guaranteed unique in scope.
> > >
> > > In fact, a lot of SICP examples have you define a function within a function with a name like "iter" for TCO recursion. It'd work like that.
> > >
> > > Edit: seems like gyka below confirms that's how it works. I should have read on!

kbp on Aug 2, 2017 [-]

> It's almost necessary for any dialect of LISP to support lambdas (i.e. functions) that contain a LISP 'do' [1] which lets you group statements.

> [1] http://www.lispworks.com/documentation/lw60/CLHS/Body/m_do_d...

'do' in most Lisps and Schemes is essentially an extended for loop, not a blocking construct. Of course, you could write

```
(do ((once nil t)) ; Scheme: (do ((once #f #t))
    (once)
  (thing-1)
  (thing-2)
  ...)
```

or something, but that's silly.

The blocking construct is in CL called PROGN and in Scheme called begin. Lambdas in CL get what's called an "implicit PROGN" around their bodies, so you can put however many forms you want inside and the value of the last is returned. It works the same in modern Scheme lambdas.

Calling PROGN/begin "do" is solely a Clojureism as far as I know.

> kazinator on Aug 2, 2017 [-]
>
> *> Calling PROGN/begin "do" is solely a Clojureism as far as I know.*
>
> Also an ANSI-CL-LOOP-ism:
>
> ```
> (loop for x in '(1 2 3) do (print x) (print x))
> 1
> 1
> 2
> 2
> 3
> 3
> ```
>
> Omit the *do* and you have a syntax error.
>
> > kbp on Aug 2, 2017 [-]
> >
> > You would have a syntax error with a single PRINT form and no DO, too.

gkya on Aug 2, 2017 [-]

IIRC in a talk the author said that it just generated a function with a made-up unique name, like an

internal def in another function.

serhei on Aug 2, 2017 [-]

That's pretty similar to how languages like Scala with higher-order features translate down to the JVM bytecode (which is subject to certain restrictions).

kazinator on Aug 2, 2017 [-]

In general, if some target language entity exists in a named form only, then the corresponding anonymity of those entities in the source language must be simulated via gensyms.

For instance, the branch targets in a while loop are anonymous, right? But in the target language, you must have a named label for the instruction. Solution: machine-generated label.

kzisme on Aug 2, 2017 [-]

Shout out to @paultag (Author/Creator of Hy) - he's an awesome guy who has written some really cool stuff.

boxy_brown on Aug 2, 2017 [-]

This, Paul is someone I look to as a model open source contributor in so many ways.

https://twitter.com/nedbat/status/861696880790704128

> How many people can say they have worked with [Sally] Yates, and have written exemplary Dockerfiles?

paultag on Aug 3, 2017 [-]

Thank you so much. I'm deeply humbled and extremely grateful for this. It's the nicest thing I've read all week, and really brightened my day up. Thank you!

srean on Aug 3, 2017 [-]

May I shower some humble upvotes sir.

Does Hy work with Pypy ? A jitter lisp would be cool

paultag on Aug 3, 2017 [-]

It does! We run tests on pypy, and it for sure works. I did at one point even test hy with rpython, I can't remember if that will still work or not, but it did compile like 4 years ago :)

srean on Aug 5, 2017 [-]

Fantastic!

paultag on Aug 3, 2017 [-]

<3

dvdt on Aug 2, 2017 [-]

One fun application: I used Hy to control my liquid handling robot,

https://twitter.com/dtsao/status/891038101706989568

gcoda on Aug 2, 2017 [-]

Reason I am trying it - best "hello world" example ever.

(print "I was going to code in Python syntax, but then I got Hy.")

nathancahill on Aug 2, 2017 [-]

This is my go-to for toy projects over the last couple years. Great execution all around. Be aware, if you don't

like puns, this isn't the language for you.

sn9 on Aug 2, 2017 [-]

So you're saying that if you don't like puns, you should get off your Hy horse?

kerkeslager on Aug 3, 2017 [-]

Nothing so aggressive. He's just saying they hold puns in Hy esteem.

souenzzo on Aug 3, 2017 [-]

See also

https://github.com/pixie-lang/pixie

nerdponx on Aug 3, 2017 [-]

I wanted to get into Pixie but last I heard it was a dead project.

souenzzo on Aug 3, 2017 [-]

It's not "dead" https://www.youtube.com/watch?v=1AjhFZVfB9c It's a weekend project, for fun. Functions that are work will not stop to work. I'm using it and I pretend to contribute when I find a problem.

nerdponx on Aug 3, 2017 [-]

Great to hear. I'm still not sold on CL as "my Lisp of choice", so I would be happy to give Pixie a shot.

kronos29296 on Aug 2, 2017 [-]

The entire thing is full of puns really too much of puns related to hy. Cool project. Python joined Erlang and Java with a lisp flavour of its own with Hy. Hope we get a let expression soon.

asimjalis on Aug 2, 2017 [-]

I have been using Hy as an alternative to Python. It is delightful to use. Highly recommended.

JadeNB on Aug 3, 2017 [-]

> using Hy …. Highly recommended.

I think that you are not getting in the proper punning spirit of the language.

cheez on Aug 2, 2017 [-]

What kind of projects have you used it for?

rcarmo on Aug 2, 2017 [-]

My blog/wiki engine is written in an older version of Hy:

https://github.com/rcarmo/sushy

I'm currently re-writing it in Python 3.6 because Hy has taken a hard stance in backwards compatibility and deprecated "let" (with a replacement macro, but still) and isn't tackling async (both for understandable, but sad reasons), so it doesn't really work for me anymore.

But while it did, I loved it to bits. Python with a LISP syntax is just wonderful, and if someone ever finds a way to add back the stuff I like, I'm more than willing to deal with the quirks.

chuckdries on Aug 2, 2017 [-]

I know python, and last semester I took a class that briefly touched (lightly kissed?) LISP for a few weeks. I have a passing familiarity and I really loved the very small amount that I saw. I'd like to learn more, can you go into more detail on the usage of let and how one would go without it?

flavio81 on Aug 4, 2017 [-]

and to directly answer your question, there a are many things you can do with let. Let binds a "form" to a symbol.

```
(let ((pi 3.14))
 ... code goes here ... )
```

So within the "code" part, that is, within that lexical environment, the symbol "pi" exists and is bound to float number 3.14

(Alternatively, you don't need to set a initial value.)

One of the interesting uses of let is that it can also redefine *dynamic variables* (aka *special* variables, think of them as "global variables).

So for example let's assume i define *timeout* to 300, so within my whole *package*, this *special variable* has the value of 300. Let's suppose this variable is going to be read by function "my-function", and others.

So i create this special variable:

```
(defparameter *timeout* 300)
```

Now, despite the above, let's suppose i want to call "my-function" but with a timeout of 1000. I can just do this in my code:

```
(let ((*timeout* 1000))
     (my-function))
```

So, inside this "let", when "my-function" gets executed, the timeout will be 1000. Outside of this, it will still be 300.

Let, thus, allows you easy use of lexical environments.

Another use is creating *closures* by using the combination of "let" with "lambda."

flavio81 on Aug 4, 2017 [-]

> I'd like to learn more, can you go into more detail on the usage of let and how one would go without it?

Welcome to the wonderful world of Lisp.

I'd recommend to you to install Portacle (the Portable Common Lisp Environment), this allows you to code in Lisp straight away.

As for tutorials, "Practical Common Lisp" (free online book) is fun, modern, and excellent for introducing you to most CL concepts, including of course use of let, let*, etc.

hood_syntax on Aug 2, 2017 [-]

> lightly kissed

How romantic!

gknoy on Aug 2, 2017 [-]

Thank you for sharing this. The last time I started trying to use Hy, I felt like I was basically writing Python but wrapped in parens -- I felt like I was doing something wrong, but couldn't place what it was, so it's nice to see a non-toy implementation that demonstrates what one can do with it.

asimjalis on Aug 2, 2017 [-]

My main use cases: Doing data analysis using Numpy, Pandas, sklearn. Calling AWS API using Boto. Calling GDAX API using their Python code samples.

I also use it for small personal scripts.

My first choice for one-off scripts would have been Clojure. But the long startup time for Clojure apps makes it unusable for quick command line scripts. Hy gives us the best of both worlds.

cutler on Aug 2, 2017 [-]

There's now Lumo - a Clojurescript binary with its own embedded Node.js runtime. Script away!

elf_m_sternberg on Aug 2, 2017 [-]

There are a few projects on my github (elfsternberg) that I've used Hy for.

One thing that I've also done is write the first draft in Hy, and then used Hy2Py to generate the version that I'm going to publish. This does result in a lot of compiling-by-hand, but the end product is often surprisingly robust.

On the other hand, it's also very (!) un-pythonic. Hy discourages classes in favor of closures, like Scheme, so I end up with a lot of nested sub-functions. My project git-linter looks like that. Hy encourages highly functional thinking, and git-linter is just that: the inputs are things like the configuration file, available linters, the command line, and (depending on the command) either the output of a "git-status --porcelain" or just "find . -type f"; the output is a report on the various outputs of the linters. It's a very straightforward map/reduce/filter, so no object-orientated code at all was required.

To an engrained Pythonista who must make a class for every last step, this land-of-verbs approach tends to look strange.

kerkeslager on Aug 3, 2017 [-]

I'm not sure I'd call the class explosions that a lot of people are writing these days Pythonic. Lots of older Python code was written with functions and a very occasional class. I think the last decade has seen an upward trend the usage of classes in Python due to the influence of Java and how software engineering is taught in colleges.

dmoney on Aug 3, 2017 [-]

Unicode identifiers make me uncomfortable. I have no idea how I would type that circular one in the "sharp macro" section (short of copying and pasting), and I'm imagining seeing those missing-character boxes all over the place.

yorwba on Aug 3, 2017 [-]

I think that example was more tongue-in-cheek than anything (after all, who would want to write their code in reverse). But if you use e.g. the fcitx input on Linux, you can just press Ctrl-Alt-Shift-U, type "circle arrow", and select "↻" (clockwise open circle arrow) from the list. The fuzzy matching is pretty awesome, I hope similar Unicode handling will come to all platforms in the future.

yablak on Aug 4, 2017 [-]

As a tensorflow dev, with several years experience writing clojure at a previous startup, I believe this is the most natural language for writing tensorflow models (or theano. basically any graph building dsl that has any real complexity and has been bolted onto python).

Suddenly TF's cond and while_loop and context controls all fit naturally into the language.

My favorite technical post in recent memory.

udkl on Aug 2, 2017 [-]

While both have their own personalities, has anyone done a side by side syntax and feature comparison of clojure and Hy ?

ealhad on Aug 2, 2017 [-]

“Keep in mind we’re not Clojure. We’re not Common Lisp. We’re Homoiconic Python, with extra bits that make sense.” — Hy Style Guide

That said, Hy takes *a lot* from Clojure.

flavio81 on Aug 3, 2017 [-]

> That said, Hy takes a lot from Clojure.

Oh oh... That scares me. Clojure -as a Lisp-like language running on the JVM- is a good idea but i have some reservations regarding how that idea got implemented.

ealhad on Aug 15, 2017 [-]

I can only agree with you. I don't like some of the syntactic choices of Clojure, and sadly some of those are in Hy.

didibus on Aug 3, 2017 [-]

Features are very different, because the features of Hy are the features of Python. So any comparison of Clojure vs Python will do.

The syntax though is very similar.

robobro on Aug 3, 2017 [-]

Has anyone ever used Hy for CGI stuff? It may be possible to port Hackernews to it, just for fun..

cuspycode on Aug 2, 2017 [-]

I used Hylang to program an Arduino Yún a few years ago, via cross-compilation. This was a very nice experience, since it allowed me to use the great Python ecosystem without having to be limited by Python the language.

sridca on Aug 3, 2017 [-]

A decade ago I wrote something like this using Racket (formerly MzScheme)! https://github.com/srid/boalisp

partingshots on Aug 2, 2017 [-]

Now CS61A can finally go back to teaching purely in Scheme again.

amelius on Aug 2, 2017 [-]

Is this like clojure, but for python instead of java?

kirbyfan64sos on Aug 4, 2017 [-]

Yup, although Hy tries to stay much closer to Python's semantics than Clojure to Java's.

droidist2 on Aug 3, 2017 [-]

How does this work with tooling though, like step debugging and refactoring in IDEs like PyCharm, or with iPython notebooks?

wodenokoto on Aug 2, 2017 [-]

Will this give me lisp in jupyther notebooks?

mmaul on Aug 2, 2017 [-]

You can already have lisp (common) in jupyter notebooks via cl-jupyter. Here is an example https://github.com/mmaul/clml.tutorials/blob/master/CLML-Win...

If you want to use Hy in a note book you will need a jupyter kernel for it and I belive there is one called hy_kernel

evanwolf on Aug 2, 2017 [-]

Next on the wishlist: an APL inside Python.

yoodenvranx on Aug 2, 2017 [-]

> Hy

This must be the worst submission title in the history of HN.

asimjalis on Aug 2, 2017 [-]

It is possibly the shortest.

hk__2 on Aug 2, 2017 [-]

I got curious and I checked; there have been a few one-character submissions so it's not the shortest.

```
⊥: https://news.ycombinator.com/item?id=7530548
{: https://news.ycombinator.com/item?id=7729399
M: https://news.ycombinator.com/item?id=4367006
```

In total (at least in the [fh-bigquery:hackernews.stories] BQ dataset), there have been 15 non-deleted submissions with a one-character title.

eanzenberg on Aug 2, 2017 [-]

[flagged]

sctb on Aug 3, 2017 [-]

Could you please don't do this here? We detached this subthread from https://news.ycombinator.com/item?id=14914448 and marked it off-topic.

parhamn on Aug 2, 2017 [-]

> equally (un)readable

Means they're the same to them regardless of whether its readable or not (they aren't even opining here).

tills13 on Aug 2, 2017 [-]

hy more like why