wisp: Whitespace to Lisp

Dr. Arne Babenhauserheide

<2013-03-26 Di>

» I love the syntax of Python, but crave the simplicity and power of Lisp.«

```
display "Hello World!"  ↦  (display "Hello World!")
```

```
define : factorial n          (define (factorial n)
    if : zero? n          ↦       (if (zero? n)
       . 1                          1
       * n : factorial {n - 1}      (* n (factorial {n - 1}))))
```

Wisp basics

- Wisp turns indentation into lisp expressions.
- Why Wisp?
- Get it
  - from its Mercurial repository:
    `hg clone https://hg.sr.ht/~arnebab/wisp`
  - Or via GNU Guix:
    `guix install guile guile-wisp`
  - Or via the package guile-wisp-hg for Arch Linux.
  - Or via `./configure; make install` from the releases.
- See more examples and tests.

»ArneBab's alternate sexp syntax is best I've seen; pythonesque, hides parens but keeps power« — Christopher Webber in twitter, in identi.ca and in his blog: Wisp: Lisp, minus the parentheses
♡ wow ♡

»Wisp allows people to see code how Lispers perceive it. Its structure becomes apparent.«
— Ricardo Wurmus in IRC, paraphrasing the wisp statement from his talk at FOSDEM 2019 about Guix for reproducible science in HPC.
◡̈ Yay! ◡̈

```
with (open-file "with.w" "r") as port
     format #t "~a\n" : read port
```

Familiar with-statement in 25 lines.

↓ skip updates and releases ↓

Update (2020-09-15): Wisp 1.0.3 provides a `wisp` binary to start a wisp repl or run wisp files, builds with **Guile 3**, and moved to sourcehut for libre hosting: hg.sr.ht/~arnebab/wisp.

After installation, just run `wisp` to enter a **wisp-shell** (REPL).

This release also ships wisp-mode 0.2.6 (fewer autoloads), ob-wisp 0.1 (initial support for org-babel), and additional examples. New auxiliary projects include wispserve for experiments with streaming and download-mesh via Guile, and wisp in conf:

```
conf new -l wisp PROJNAME
```

creates an **autotools project with wisp**, while

```
conf new -l wisp-enter PROJAME
```

creates a project with natural script writing and guile doctests set up. Both also install a script to run your project with minimal start time: I see 23ms to 27ms runtime for hello world with rare outliers at 100ms. The name of the script is the name of your project.

For more info about Wisp 1.0.3, see the NEWS file.
To test wisp v1.0.3, install Guile 2.0.11 or later and bootstrap wisp:

```
wget https://www.draketo.de/files/wisp-1.0.3.tar_.gz; \
  tar xf wisp-1.0.3.tar_.gz ; cd wisp-1.0.3/; \
  ./configure; make check; examples/newbase60.w 123
```

If it prints 23 (123 in NewBase60), your wisp is fully operational.
If you have additional questions, see the Frequently asked Questions (FAQ) and chat in #guile at freenode.
That's it - have fun with wisp syntax!

Update (2019-07-16): wisp-mode 0.2.5 now provides proper indentation support in Emacs: *Tab* increases indentation and cycles back to zero. *Shift-tab* decreases indentation via previously defined indentation levels. *Return* preserves the indentation level (hit tab twice to go to zero indentation).

Update (2019-06-16): In c programming the uncommon way, specifically c-indent, tantalum is experimenting with combining wisp and sph-sc, which compiles scheme-like s-expressions to c. The result is a program written like this:

```
pre-include "stdio.h"

define (main argc argv) : int int char**
  declare i int
  printf "the number of arguments is %d\n" argc
  for : (set i 0) (< i argc) (set+ i 1)
    printf "arg %d is %s\n" (+ i 1) (array-get argv i)
  return 0 ;; code-snippet under GPLv3+
```

To me that looks so close to C that it took me a moment to realize that it isn't just using a parser which allows omitting some special syntax of C, but actually an implementation of a C-generator in Scheme (similar in spirit to cython, which generates C from Python), which results in code that looks like a more regular version of C without superfluous parens. Wisp really completes the round-trip from C over Scheme to *something that looks like C but has all the regularity of Scheme*, because all things considered, the code example is regular wisp-code. And it is awesome to see tantalum take up the tool I created and use it to experiment with ways to program that I never even imagined! ♡

TLDR: tantalum uses wisp for code that looks like C and compiles to C but has the regularity of Scheme!

Update (2019-06-02): The repository at https://www.draketo.de/proj/wisp/ is stale at the moment, because the staticsite extension I use to update it was broken by API changes and I currently don't have the time to fix it. Therefore until I get it fixed, the canonical repository for wisp is `https://bitbucket.org/ArneBab/wisp/`. I'm sorry for that. I would prefer to self-host it again, but the time to read up what i have to adjust blocks that right now (typically the actual fix only needs a few lines). A pull-request which fixes the staticsite extension for modern Mercurial would be **much appreciated**!

Update (2019-02-08): wisp v1.0 released as announced at FOSDEM. Wisp the language is complete:

```
display "Hello World!"      ↦ (display "Hello World!")
```

And it achieves its goal:

"Wisp allows people to see code how Lispers perceive it. Its structure becomes apparent."
— Ricardo Wurmus at FOSDEM

Tooling, documentation, and porting of wisp are still work in progress, but before I go on, I want thank the people from the readable lisp project. Without our initial shared path, and without their encouragement, wisp would not be here today. Thank you! You're awesome!

With this release it is time to put wisp to use. To start your own project, see the tutorial **Starting a wisp project** and the wisp tutorial. For more info, see the NEWS file. To test wisp v1.0, install Guile 2.0.11 or later and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-1.0.tar.gz; \
  tar xf wisp-1.0.tar.gz ; cd wisp-1.0/; \
  ./configure; make check; examples/newbase60.w 123
```

If it prints 23 (123 in NewBase60), your wisp is fully operational.
If you have additional questions, see the Frequently asked Questions (FAQ) and chat in #guile at freenode.
That's it - have fun with wisp syntax!

Update (2019-01-27): wisp v0.9.9.1 released which includes the emacs support files missed in v0.9.9, but excludes unnecessary files which increased the release size from 500k to 9 MiB (it's now back at about 500k). To start your own wisp-project, see the tutorial **Starting a wisp project** and the wisp tutorial. For more info, see the NEWS file. To test wisp v0.9.9.1, install Guile 2.0.11 or later and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.9.9.1.tar.gz; \
  tar xf wisp-0.9.9.1.tar.gz ; cd wisp-0.9.9.1/; \
  ./configure; make check; examples/newbase60.w 123
```

If it prints 23 (123 in NewBase60), your wisp is fully operational.
That's it - have fun with wisp syntax!

Update (2019-01-22): wisp v0.9.9 released with support for literal arrays in Guile (needed for doctests), example start times below 100ms, ob-wisp.el for emacs org-mode babel and work on examples: network, securepassword, and downloadmesh. To start your own wisp-project, see the tutorial **Starting a wisp project** and the wisp tutorial. For more info, see the NEWS file. To test wisp v0.9.9, install Guile 2.0.11 or later and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.9.9.tar.gz; \
  tar xf wisp-0.9.9.tar.gz ; cd wisp-0.9.9/; \
  ./configure; make check; examples/newbase60.w 123
```

If it prints 23 (123 in NewBase60), your wisp is fully operational.
That's it - have fun with wisp syntax!

Update (2018-06-26): There is now a wisp tutorial for beginning programmers: *"In this tutorial you will learn to write programs with wisp. It requires no prior knowledge of programming."* — Learn to program with Wisp, published in With Guise and Guile

Update (2017-11-10): wisp v0.9.8 released with installation fixes (thanks to benq!). To start your own wisp-project, see the tutorial **Starting a wisp project**. For more info,

see the NEWS file. To test wisp v0.9.8, install Guile 2.0.11 or later and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.9.8.tar.gz; \
  tar xf wisp-0.9.8.tar.gz ; cd wisp-0.9.8/; \
  ./configure; make check; examples/newbase60.w 123
```

If it prints 23 (123 in NewBase60), your wisp is fully operational.
That's it - have fun with wisp syntax!

Update (2017-10-17): wisp v0.9.7 released with bugfixes. To start your own wisp-project, see the tutorial **Starting a wisp project**. For more info, see the NEWS file. To test wisp v0.9.7, install Guile 2.0.11 or later and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.9.7.tar.gz; \
  tar xf wisp-0.9.7.tar.gz ; cd wisp-0.9.7/; \
  ./configure; make check; examples/newbase60.w 123
```

If it prints 23 (123 in NewBase60), your wisp is fully operational.
That's it - have fun with wisp syntax!

Update (2017-10-08): wisp v0.9.6 released with compatibility for tests on OSX and old autotools, installation to `guile/site/(guile version)/language/wisp` for cleaner installation, debugging and warning when using not yet defined lower indentation levels, and with `wisp-scheme.scm` moved to `language/wisp.scm`. This allows creating a wisp project by simply copying `language/`. A short tutorial for creating a wisp project is available at **Starting a wisp project** as part of With Guise and Guile. For more info, see the NEWS file. To test wisp v0.9.6, install Guile 2.0.11 or later and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.9.6.tar.gz; \
  tar xf wisp-0.9.6.tar.gz ; cd wisp-0.9.6/; \
  ./configure; make check; examples/newbase60.w 123
```

If it prints 23 (123 in NewBase60), your wisp is fully operational.
That's it - have fun with wisp syntax!

Update (2017-08-19): Thanks to tantalum, wisp is now available as package for Arch Linux: from the Arch User Repository (AUR) as guile-wisp-hg! Instructions for installing the package are provided on the AUR page in the Arch Linux wiki. Thank you, tantalum!

Update (2017-08-20): wisp v0.9.2 released with many additional examples including the proof-of-concept for a minimum ceremony dialog-based game duel.w and the datatype benchmarks in benchmark.w. For more info, see the NEWS file. To test it, install Guile 2.0.11 or later and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.9.2.tar.gz; \
```

```
tar xf wisp-0.9.2.tar.gz ; cd wisp-0.9.2/; \
./configure; make check; examples/newbase60.w 123
```

If it prints 23 (123 in NewBase60), your wisp is fully operational.
That's it - have fun with wisp syntax!

Update (2017-03-18): I removed the link to Gozala's wisp, because it was put in maintenance mode. Quite the opposite of Guile which is taking up speed and just released Guile version 2.2.0, fully compatible with wisp (though wisp helped to find and fix one compiler bug, which is something I'm really happy about ◡̈).

Update (2017-02-05): Allan C. Webber presented my talk Natural script writing with Guile in the Guile devroom at FOSDEM. The talk was awesome — and recorded! Enjoy Natural script writing with Guile by "pretend Arne" ◡̈
Download the video. Get the presentation (pdf, 16 slides) and its source (org).
Have fun with wisp syntax!

Update (2016-07-12): wisp v0.9.1 released with a fix for multiline strings and many additional examples. For more info, see the NEWS file. To test it, install Guile 2.0.11 or later and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.9.1.tar.gz; \
  tar xf wisp-0.9.1.tar.gz ; cd wisp-0.9.1/; \
  ./configure; make check; examples/newbase60.w 123
```

If it prints 23 (123 in NewBase60), your wisp is fully operational.
That's it - have fun with wisp syntax!

Update (2016-01-30): I presented Wisp in the Guile devroom at FOSDEM. The reception was unexpectedly positive — given some of the backlash the readable project got I expected an exceptionally sceptical audience, but people rather asked about ways to put Wisp to good use, for example in templates, whether it works in the REPL (yes, it does) and whether it could help people start into Scheme.

> *Wisp is "The power and simplicity of #lisp with the familiar syntax of #python" talk by @ArneBab #fosdem pic.twitter.com/TaGhIGruIU* — Jan Nieuwenhuizen (@JANieuwenhuizen) January 30, 2016

The atmosphere in the Guile devroom was very constructive and friendly during all talks, and I'm happy I could meet the Hackers there in person. I'm definitely taking good memories with me. Sadly the video did not make it, but the schedule-page includes the presentation (pdf, 10 slides) and its source (org).
Have fun with wisp syntax!

Update (2016-01-04): Wisp is available in GNU Guix! Thanks to the package from Christopher Webber you can try Wisp easily on top of any distribution:

```
guix package -i guile guile-wispguile --language=wisp
```

This already gives you Wisp at the REPL (take care to follow all instructions for installing Guix on top of another distro, especially the locales).
Have fun with wisp syntax!

Update (2015-10-01): wisp v0.9.0 released which no longer depends on Python for bootstrapping releases (but ./configure still asks for it — a fix for another day). And thanks to Christopher Webber there is now a patch to install wisp within GNU Guix. For more info, see the NEWS file. To test it, install Guile 2.0.11 or later and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.9.0.tar.gz; \
  tar xf wisp-0.9.0.tar.gz ; cd wisp-0.9.0/; \
  ./configure; make check; examples/newbase60.w 123
```

If it prints 23 (123 in NewBase60), your wisp is fully operational.
That's it - have fun with wisp syntax!

Update (2015-09-12): wisp v0.8.6 released with fixed macros in interpreted code, chunking by top-level forms, `:  .` parsed as nothing, ending chunks with a trailing period, updated example evolve and added examples newbase60, cli, cholesky decomposition, closure and hoist in loop. For more info, see the NEWS file.To test it, install Guile 2.0.x or 2.2.x and Python 3 and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.8.6.tar.gz; \
  tar xf wisp-0.8.6.tar.gz ; cd wisp-0.8.6/; \
  ./configure; make check; examples/newbase60.w 123
```

If it prints 23 (123 in NewBase60), your wisp is fully operational.
That's it - have fun with wisp syntax! And a happy time together for the ones who merge their paths today ⌣̈

Update (2015-04-10): wisp v0.8.3 released with line information in backtraces. For more info, see the NEWS file.To test it, install Guile 2.0.x or 2.2.x and Python 3 and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.8.3.tar.gz; \
  tar xf wisp-0.8.3.tar.gz ; cd wisp-0.8.3/; \
  ./configure; make check; guile -L . --language=wisp tests/factorial.w; echo
```

If it prints 120120 (two times 120, the factorial of 5), your wisp is fully operational.
That's it - have fun with wisp syntax!

Update (2015-03-18): wisp v0.8.2 released with reader bugfixes, new examples and an updated draft for SRFI 119 (wisp). For more info, see the NEWS file.To test it, install

Guile 2.0.x or 2.2.x and Python 3 and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.8.2.tar.gz; \
  tar xf wisp-0.8.2.tar.gz ; cd wisp-0.8.2/; \
  ./configure; make check; guile -L . --language=wisp tests/factorial.w; echo
```

If it prints 120120 (two times 120, the factorial of 5), your wisp is fully operational. That's it - have fun with wisp syntax!

Update (2015-02-03): The wisp SRFI just got into draft state: SRFI-119 — on its way to an official Scheme Request For Implementation!

Update (2014-11-19): wisp v0.8.1 released with reader bugfixes. To test it, install Guile 2.0.x and Python 3 and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.8.1.tar.gz; \
  tar xf wisp-0.8.1.tar.gz ; cd wisp-0.8.1/; \
  ./configure; make check; guile -L . --language=wisp tests/factorial.w; echo
```

If it prints 120120 (two times 120, the factorial of 5), your wisp is fully operational. That's it - have fun with wisp syntax!

Update (2014-11-06): wisp v0.8.0 released! The new parser now passes the testsuite and wisp files can be executed directly. For more details, see the NEWS file. To test it, install Guile 2.0.x and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.8.0.tar.gz; \
  tar xf wisp-0.8.0.tar.gz ; cd wisp-0.8.0/; \
  ./configure; make check; guile -L . --language=wisp tests/factorial.w; echo
```

If it prints 120120 (two times 120, the factorial of 5), your wisp is fully operational. That's it - have fun with wisp syntax!

On a personal note: It's mindboggling that I could get this far! This is actually a fully bootstrapped indentation sensitive programming language with all the power of Scheme underneath, and it's a one-person when-my-wife-and-children-sleep sideproject. The extensibility of Guile is awesome!

Update (2014-10-17): wisp v0.6.6 has a new implementation of the parser which now uses the scheme read function. 'wisp-scheme.w' parses directly to a scheme syntax-tree instead of a scheme file to be more suitable to an SRFI. For more details, see the NEWS file. To test it, install Guile 2.0.x and bootstrap wisp:

```
wget https://bitbucket.org/ArneBab/wisp/downloads/wisp-0.6.6.tar.gz; \
  tar xf wisp-0.6.6.tar.gz; cd wisp-0.6.6; \
  ./configure; make; guile -L . --language=wisp
```

That's it - have fun with wisp syntax at the REPL!
Caveat: It does not support the ' prefix yet (syntax point 4).

Update (2014-01-04): Resolved the name-clash together with Steve Purcell und Kris Jenkins: the javascript wisp-mode was renamed to wispjs-mode and wisp.el is called wisp-mode 0.1.5 again. It provides syntax highlighting for Emacs and minimal indentation support via tab. You can install it with 'M-x package-install wisp-mode'

Update (2014-01-03): wisp-mode.el was renamed to wisp 0.1.4 to avoid a name clash with wisp-mode for the javascript-based wisp.

Update (2013-09-13): Wisp now has a REPL! Thanks go to GNU Guile and especially Mark Weaver, who guided me through the process (along with nalaginrut who answered my first clueless questions...).
To test the REPL, get the current code snapshot, unpack it, run `./bootstrap.sh`, start guile with `$ guile -L .` (requires guile 2.x) and enter `,language wisp`.
Example usage:

```
display "Hello World!\n"
```

then hit enter thrice.
Voilà, you have wisp at the REPL!
/Caveeat: the wisp-parser is still experimental and contains known bugs. Use it for testing, but please do not rely on it for important stuff, yet./

Update (2013-09-10): wisp-guile.w can now parse itself! Bootstrapping: The magical feeling of seeing a language (dialect) grow up to live by itself: `python3 wisp.py wisp-guile.w > 1 && guile 1 wisp-guile.w > 2 && guile 2 wisp-guile.w > 3 && diff 2 3`. Starting today, wisp is implemented in wisp.

Update (2013-08-08): Wisp 0.3.1 released (Changelog).

↑ to most recent updates ↑

Contents

What is wisp?

Wisp is a simple preprocessor that turns indentation sensitive syntax into Lisp syntax.

The goal is to create the **simplest possible** indentation based syntax which is able to express **all possibilities of Lisp**.

It works by inferring the parentheses of lisp by reading the indentation of lines.

Wisp is related to SRFI-49 and the readable Lisp S-expressions Project (and actually inspired by the latter), but it tries to *Keep it Simple and Stupid*: wisp is a simple preprocessor which can be called by any lisp implementation to add support for indentation sensitive syntax. To repeat the initial quote:

I love the syntax of Python, but crave the simplicity and power of Lisp.

With wisp I want to make it possible to create lisp code which is easily **readable for non-programmers** (and me!) and at the same time keeps the **simplicity and power of Lisp**.

Its main **technical improvement** over SRFI-49 and Project Readable is using lines prefixed by a dot (". ") to mark the continuation of the parameters of a function after intermediate function calls.

The dot-syntax means that instead of marking every function call, Wisp marks every line which does *not* begin with a function call — which is the much less common case in lisp-code.

See the basics and releases for information how to get the current version of wisp.

Frequently asked Questions

- Can this represent any Scheme code? Yes. Wisp enables you to write arbitrary code structures using indentation. When you write code in wisp and run it with Guile, it is full Scheme code with all its capabilities.
- How do Macros work with wisp? Just like they work in Scheme code that has parentheses: Write the same structure as with Scheme but use indentation for structure instead of parentheses where that is more readable to you or your future readers. See for example the macro-writing-macro `Enter` in Enter three witches.

Wisp syntax rules

1. **A line without indentation is a function call**, just as if it would start with a parenthesis.

   ```
   display "Hello World!"      ↦      (display "Hello World!")
   ```

2. **A line which is more indented than the previous line is a sibling to that line**: It opens a new parenthesis.

   ```
   display                            ↦ (display
     string-append "Hello " "World!" ↦   (string-append "Hello " "World!"))
   ```

3. **A line which is not more indented than previous line(s) end of all previous lines which have higher or equal indentation**. You should only reduce the indentation to indentation levels which were already used by parent lines, else the behaviour is undefined.

```
display                            ↦ (display
  string-append "Hello " "World!" ↦   (string-append "Hello " "World!"))
display "Hello Again!"             ↦ (display "Hello Again!")
```

4. **To add any of ' , or ‘ to a line, just prefix the line** with any combination of "' ", ", " or "‘ " (symbol followed by one space).

```
' "Hello World!"      ↦      '("Hello World!")
```

5. **A line whose first non-whitespace characters are a dot followed by a space (". ") does not call a function: it is treated as simple continuation of the first less indented previous line**. In the first line this means that this line does not start with a parenthesis and does not end with a parenthesis, just as if you had directly written it in lisp without the leading ". ".

```
string-append "Hello"         ↦    (string-append "Hello"
  string-append " " "World"   ↦      (string-append " " "World")
  . "!"                       ↦      "!")
```

6. **A line which contains only whitespace and a colon (":") defines an indentation level at the indentation of the colon**. It opens a parenthesis which gets closed by the next less- or equal-indented line. To use an actual colon. you can escape it as "\:".

```
let                       ↦    (let
  :                       ↦      ((msg "Hello World!"))
    msg "Hello World!"    ↦      (display msg))
  display msg             ↦
```

7. **A colon sourrounded by whitespace (" : ") in a non-empty line starts a new sub-line which gets closed at the end of the line**. It is an inline parenthesis.

```
define : hello who                 ↦ (define (hello who)
  display                          ↦   (display
    string-append "Hello " who "!" ↦     (string-append "Hello " who "!")))
```

8. **You can replace any number of consecutive initial spaces by underscores**, as long as at least one whitespace is left between the underscores and any following character. You can escape initial underscores by prefixing the first one with \ ("__a" → "(_a)"), *if you have to use them as function names.*

```
define : hello who                  ↦ (define (hello who)
_ display                           ↦   (display
___ string-append "Hello " who "!" ↦     (string-append "Hello " who "!")))
```

To make that easier to understand, let's just look at the examples in more detail:

1 A simple top-level function call

```
display "Hello World!"      ↦      (display "Hello World!")
```

This one is easy: Just add a bracket before and after the content.

2 Nested function calls

```
display                             ↦ (display
  string-append "Hello " "World!" ↦   (string-append "Hello " "World!"))
```

If a line is more indented than a previous line, it is a sibling to the previous function: The brackets of the previous function gets closed after the (last) sibling line.

3 Multiple function calls

```
display "Hello World!"      ↦      (display "Hello World!")
display "Hello Again!"      ↦      (display "Hello Again!")
```

Multiple lines with the same indentation are separate function calls (except if one of them starts with ". ", see , shown in a few lines).

4 Wrap a line in a special form

```
' "Hello World!"      ↦      '("Hello World!")
```

The special forms `’ ‘ , @, #’ #‘ #, #,@` can used as wrapper for a whole line by prefixing the line with them.

5 Continue function arguments

By using a . followed by a space as the first non-whitespace character on a line, you can mark it as continuation of the previous less-indented line. Then it is no function call but continues the list of parameters of the funtcion.

I use a very synthetic example here to avoid introducing additional unrelated concepts.

```
string-append "Hello"        ↦    (string-append "Hello"
  string-append " " "World"  ↦      (string-append " " "World")
  . "!"                      ↦      "!")
```

As you can see, the final "!" is not treated as a function call but as parameter to the first string-append.

This syntax extends the notion of the dot as identity function. In many lisp implementations we already have ‘(= a (. a))‘.

```
= a        ↦    (= a
  . a      ↦      (. a))
```

With wisp, we extend that equality to ‘(= ’(a b c) ’((. a b c)))‘.

```
. a b c    ↦    a b c
```

6 Double parens (let-notation)

If you use ‘let‘, you often need double parentheses. Since using pure indentation in empty lines would be really error-prone, we need a way to mark a line as indentation level.

To add multiple parentheses, we use a colon to mark an intermediate line as additional indentation level.

```
let                       ↦    (let
  :                       ↦      ((msg "Hello World!"))
    msg "Hello World!"    ↦      (display msg))
  display msg             ↦
```

7 One-line function calls inline

Since we already use the colon as syntax element, we can make it possible to use it everywhere to open a parenthesis - even within a line containing other code. Since wide unicode characters would make it hard to find the indentation of that colon, such an inline-function call always ends at the end of the line. Practically that means, the opened parenthesis of an inline colon always gets closed at the end of the line.

```
define : hello who                          ↦ (define (hello who)
  display : string-append "Hello " who "!" ↦   (display (string-append "Hello " who
```

This also allows using inline-let:

```
let                          ↦    (let
  : msg "Hello World!"       ↦      ((msg "Hello World!"))
  display msg                ↦      (display msg))
```

and can be stacked for more compact code:

```
let : : msg "Hello World!" ↦ (let ((msg "Hello World!"))
  display msg              ↦   (display msg))
```

8 Visible indentation

To make the indentation visible in non-whitespace-preserving environments like badly written html, you can replace any number of consecutive initial spaces by underscores, as long as at least one whitespace is left between the underscores and any following character. You can escape initial underscores by prefixing the first one with \ ("_ _a" → "(_a)"), *if you have to use them as function* names.

```
define : hello who                    ↦    (define (hello who)
_ display                             ↦      (display
___ string-append "Hello " who "!"    ↦        (string-append "Hello " who "!")))
```

Syntax justification

I do not like adding any unnecessary syntax element to lisp. So I want to show explicitely why the syntax elements are required to meet the goal of wisp: indentation-based lisp with a simple preprocessor.

4.1 . (the dot)

We have to be able to continue the arguments of a function after a call to a function, and we must be able to split the arguments over multiple lines. That's what the leading dot allows. Also the dot at the beginning of the line as marker of the continuation of a variable list is a generalization of using the dot as identity function - which is an implementation detail in many lisps.

'(. a)' is just 'a'.

So for the single variable case, this would not even need additional parsing: wisp could just parse ". a" to "(. a)" and produce the correct result in most lisps. But forcing programmers to always use separate lines for each parameter would be very inconvenient, so the definition of the dot at the beginning of the line is extended to mean "take every element in this line as parameter to the parent function".

Essentially this dot-rule means that we mark variables at the beginning of lines instead of marking function calls, since in Lisp variables at the beginning of a line are much rarer than in other programming languages. In Lisp, assigning a value to a variable is a function call while it is a syntax element in many other languages. What would be a variable at the beginning of a line in other languages is a function call in Lisp.

(Optimize for the common case, not for the rare case)

4.2 : (the colon)

For double parentheses and for some other cases we must have a way to mark indentation levels without any code. I chose the colon, because it is the most common non-alpha-numeric character in normal prose which is not already reserved as syntax by lisp when it is surrounded by whitespace, and because it already gets used for marking keyword arguments to functions in Emacs Lisp, so it does not add completely alien characters.

The function call via inline " : " is a limited generalization of using the colon to mark an indentation level: If we add a syntax-element, we should use it as widely as possible to justify the added syntax overhead.

But if you need to use : as variable or function name, you can still do that by escaping it with a backslash (example: "\:"), so this does not forbid using the character.

4.3 _ (the underscore)

In Python the whitespace hostile html already presents problems with sharing code - for example in email list archives and forums. But in Python the indentation can mostly be inferred by looking at the previous line: If that ends with a colon, the next line must be more indented (there is nothing to clearly mark reduced indentation, though). In wisp we do not have this help, so we need a way to survive in that hostile environment.

The underscore is commonly used to denote a space in URLs, where spaces are inconvenient, but it is rarely used in lisp (where the dash ("-") is mostly used instead), so it seems like a a natural choice.

You can still use underscores anywhere but at the beginning of the line. If you want to use it at the beginning of the line you can simply escape it by prefixing the first underscore with a backslash (example: "_ _").

Background

Around the fall of 2012 I found the readable Lisp project which aims at producing indentation based lisp, and I was thrilled. I had already done a small experiment with an indentation to lisp parser, but I was more than willing to throw out my crappy code for the well-integrated parser they had.

Fast forward half a year. It's February 2013 and I started reading the readable list again after being out of touch for a few months because the birth of my daughter left little time for side-projects. And I was shocked to see that the readable folks had piled lots of additional syntax elements on their beautiful core model, which for me destroyed the simplicity and beauty of lisp. When language programmers add ***syntax using \ $ and <>****, you can be sure that it is* ***no simple lisp anymore****. To me readability does not just mean beautiful code, but rather easy to understand code with simple concepts which are used consistently. I prefer having some ugly corner cases to adding more syntax which makes the whole language more complex.*

I told them about that and proposed a simpler structure which achieved almost the same as their complex structure. To my horror they proposed ***adding*** *my proposal to readable, making it even more bloated (in my opinion). We discussed a long time - the current syntax for inline-colons is a direct result of that discussion in the readable list — then Alan wrote me a nice email, explaining that readable will keep its direction. He finished with* «We hope you continue to work with or on indentation-based syntaxes for Lisp, whether sweet-expressions, your current proposal, or some other future notation you can develop.»

It took me about a month to answer him, but the thought never left my mind (@Alan: See what you did? You anchored the thought of indentation based lisp even deeper in my mind. As if I did not already have too many side-projects... :)).

Then I had finished the first version of a simple whitespace-to-lisp preprocessor.

And today I added support for reading indentation based lisp from standard input which allows actually using it as in-process preprocessor without needing temporary files, so I think it is ***time for a real release*** *outside my Mercurial repository.*

So: Have fun with wisp v0.2 (tarball)!

PS: Wisp is linked in the comparisions of SRFI-110.

See also

Users of Wisp

- *Guix Workflow Language Reference Manual — with lots of examples using wisp*
- *Dryads Wake — the beginning of a game in a low ceremony embedded language with Wisp*

Using Wisp

- *Learn to program with Wisp — a tutorial*

- *Small Snippets with Wisp*
- *Code Katas in Scheme with Wisp*
- *Starting a wisp project — getting the infrastructure going, see conf for an easier way*

About Wisp and Scheme

- *Live stream from the Guile devroom at FOSDEM 2017!*
- *Using Guile Scheme Wisp for low ceremony embedded languages*
- *With Guise and Guile — (Sharing the Art of) Elegant and Efficient Code*
- *Recursion wins!*
- *Let-Rekursion ist toll! (German)*

Discussion

Sylvain Benner

Sylvain Benner gave constructive criticism on wisp which led to an interesting discussion and explanations of the reasons behind the design choices of wisp. He allowed me to share it here, too.

Sylvain Benner - 26.03.2013

I find it less readable and actually it adds complexity to the syntax because you have to deal with 3 components to understand the structure:

- *the new lines*
- *the colon*
- *the use of parentheses (in case of a multiple lines sexp due to line length)*

So in fact you add 2 more constructs for readability which IMHO has not the expected effect on most people. Python is more readable because it simplify the constructs over C and alike, the thing is lisp is already dead simple syntax. Finally lisp is about lists and the python syntax just make it feel less closer this.

Arne Babenhauserheide - 26.03.2013

The main problem I see with lisp syntax for newcomers is that the most important character for recognizing an expression in text is the first character[1]. And in lisp that is always the same: (

Also familiarity has a big impact on readability. And from my understanding, familiarity means a letter distribution which is similar to normal text.

Lisp actually is the best language when it comes to that: the basic constructs .,'' are the most common non-ascii-characters in normal text. But sadly the most common symmetric characters () are still very rare in normal text when compared to lisp. And those are in the most prominent place in Lisp: At the beginning of almost every line. And that makes lisp code look alien.

Those two reasons are why I kept working on indentation-sensitive syntax after taking the time to actually think it through for myself - and trying to find out why Python seemed so familiar from the start.

[1]: I'm sure you can find linguistic research for that. I can offer an anecdote: I'm a roleplayer which means that I often invent and portray different people. From my experience there is one sure way to make the others mix up those different people: Give them names with the same first letter.

Sylvain Benner - 26.03.2013

Oh you can use the dot for line continuation :) Your syntax is a smart exercise and congrats for this achievement but it is not a good thing to use in the long run, learning a lisp with this syntax for new comers is not the best idea.

Arne Babenhauserheide - 26.03.2013

I disagree: I obviously think that it's a good thing if people use indentation based lisp, why else should I write the preprocessor? :)

You might need parentheses if you want to reduce the indentation in continuation because that could actually enhance readability in some situations. This might be a fringe case (which is why I don't mind if it's a bit ugly), but it might crop up.

My main question is: Does the syntax manage to capture all features of lisp - is it a complete representation?

If that is true, then you should have all the power of lisp and the representation would be a purely aesthetic aspect (which is important, because it affects how the code feels, but it would then not be a limiting factor).

Sylvain Benner - 26.03.2013

Well, I gave it a quick try and 2 feedbacks:

- *the colon does not feel right for me, I would prefer a inlined colon and 2 levels of indentation beneath for the variable bindings list. In case of 1 list you would still be able to put it inline.*
- *with a syntax highlighter it would be cool to colorize or bold the function calls. Having electric indentation would be cool. Maybe it is possible to achieve paredit by wrapping it (transform to lisp, apply paredit, transform to wisp).*

Arne Babenhauserheide - 26.03.2013

Thanks for testing!

How would the inline colon look in code? (example)

syntax highlighting would really be cool, but before going that way, the syntax itself has to be right.

Sylvain Benner - 03:24

do you think it would make sense to drop the colon and keep only indentations to define the nested level, so only the following construct would be accepted ?

```
let
    var1 value1
    var2 value2
  dostuff var1 var2
```

Arne Babenhauserheide - 09:08

The colon cannot be dropped completely, because it is required when you have empty parentheses or two blocks with double parentheses. Use a simple defun or imagine a double let without action[1] (I'm sure you'll find real code which fits better for this example):

```
(defun foo ()
  bar)

(doublelet
  ((foo bar))
  ((bla foo)))
```

The wisp version of this is

```
defun foo
```

```
  :
  . bar

doublelet
  :
    foo bar
  : ; <- this double backstep is the real issue
    bla foo
```

or shorter with inline colon (which you can only use if there are no not-tail-called functions inside the assignments).

```
defun foo :
  . bar

doublelet
  : foo bar
  : bla foo
```

The need to be able to represent things like this is the real reason, why the colon exists. The inline and start-of-line use is only a generalization of that principle (we add that syntax, so we should see how far we can push it to reduce the effective cost of introducing the additional syntax).

[1]: I used a double let without action, even though that does nothing, because that makes it impossible to use later indentation. Another reason why I would not use later indentation to define whether something earlier is a single or double indent is that this would call for subtle and really hard to find errors:

```
defun flubb

    nubb
   gam
```

would become

```
(defun flubb ()
   ((nubb))
  (gam))
```

Also I think that fixed indentation width (alternative option to inferring it from later lines) would make it really hard to write readable code. Stuff like this would not be possible:

```
if
    equals wrong
           isright? stuff
    dostuff
```

Arne Babenhauserheide - 09:10

As syntax wisp has to be able to represent every lisp construct you can think of (to avoid being just a leaky abstraction where the writers still have to use parentheses at times). Not every fringe case has to be beautiful, but it must be possible to represent it.

Arne Babenhauserheide - 19:42

Can I copy our discussion here into the comment section of the article (under GPL)? I think it could interest others, too.

Sylvain Benner - 03:36

Yes you can.

Arne Babenhauserheide - 10:03

Thanks!