Wisp - SRFI-119

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

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

Why Wisp?

(Hello World!)

Why Wisp?

¥Hello World!£

Why Wisp?

Hello World!

Why Wisp?

Hello World!

- *The first and last letter are important for word recognition.*[1]
- *Over 70% of the codelines in the Guile scheme source start with a paren ⇒ ceremony.*
- *Many people avoid Lisp-like languages because of the parens.*[2]

[1]: *Though not all-important. See www.mrc-cbu.cam.ac.uk/people/matt.davis/cmabridge/*

[2]: *Also see srfi.schemers.org/srfi-110/srfi-110.html#cant-improve*

The most common letters: Lisp and Scheme are awesome

. , " : ’ _#? ! ;

The most common non-letter, non-math characters in prose[1]

( )

The most common paired characters[1]

[1]: From letter distributions in newspapers, see:
bitbucket.org/ArneBab/evolve-keyboard-layout/src/tip/1-gramme.arne.txt

Wisp in a nutshell

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

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

- indent as with parens, dot-prefix, inline-:, and use SRFI-105.
- Wisp uses the minimal syntax required to represent arbitrary structure:
  Syntax justification: draketo.de/english/wisp#sec-4
- Many more examples in “From Python to Guile Scheme”:
  info: draketo.de/py2guile
  download: draketo.de/proj/py2guile/py2guile.pdf

Implementation

REPL and Reader (language wisp spec)

```
define-language wisp
  . #:title "Wisp Scheme Syntax.."
  . #:reader read-one-wisp-sexp
  . #:compilers `(
      (tree-il . ,compile-tree-il))
  . #:decompilers `(
      (tree-il . ,decompile-tree-il))
  . #:evaluator (lambda (x module)
                   primitive-eval x)
  . #:printer write
  . #:make-default-environment
  lambda :
    let : : m : make-fresh-user-module
      module-define! m 'current-reader
                        make-fluid
      module-set! m 'format simple-format
      . m
```

Preprocessor (wisp.scm)

```
guile wisp.scm tests/hello.w
```

```
(define (hello who)
  (format #t "~A ~A!\n"
          "Hello" who))
(hello "Wisp")
```

(Plan B: You can always go back)

Applications?

Example: User Scripts

```
Enter : First_Witch
        Second_Witch
        Third_Witch

First_Witch
  When shall we three meet again
  In thunder, lightning, or in rain?
```

This displays

```
First Witch
  When shall we three meet again
  In thunder, lightning, or in rain?
```

- draketo.de/english/wisp/shakespeare
- Templates, executable pseudocode, REPL-interaction, configuration, . . .

Solutions

Run examples/newbase60.w as script

```
#!/usr/bin/env sh
# -*- wisp -*-
exec guile -L $(dirname $(dirname $(realpath "$0"))) --language=wisp \
           -e '(@@ (examples newbase60) main)' \
           -s "$0" "$@"
; !#
define-module : examples newbase60

define : main args
  ...
```

Use Wisp code from parenthesized Scheme

- precompile: `guile --language=wisp module`
- then just import as usual: `(use-modules (...))`

Experience

*»ArneBab's alternate sexp syntax is best I've seen; pythonesque, hides parens but keeps power« — Christopher Webber
→ dustycloud.org/blog/wisp-lisp-alternative/*

- Wisp is implemented in Wisp (850 lines, implementations).
- Examples: 4 lines (factorial) to 330 lines (advection on icosaheder).

Try Wisp

Install

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

- http://draketo.de/english/wisp

Emacs mode for syntax highlighting

- M-x package-install [RET] wisp-mode [RET]
- https://marmalade-repo.org/packages/wisp-mode

Thank you!

Why not SRFI-110 or SRFI-49?

SRFI-49

```
+ 5
  * 4 3
  2
  1
  0
```

- Cannot continue the argument list

Wisp

```
+ 5
  * 4 3
  . 2 1 0
```

SRFI-110

```
myfunction
  x: \\ original-x
  y: \\ calculate-y original-y
```

```
a b $ c d e $ f g
```

```
let <* x getx() \\ y gety() *>
! {{x * x} + {y * y}}
```

- most common letters?

Keep parens where they help readability

```
cond
  : and (null? l) (zero? a)
    . '()
  else
    cons a l
```

```
map
  lambda (x) (+ x 1)
  list 1 2 3
```