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

Started 2013.
Spec in SRFI-119 since 2015.
It's time for 1.0.

Who am I

- Python since 2006, from 2011 to 2017 for my PhD in Physics, along with Fortran, who guessed it? :-)
- Using Emacs since 2008, contributed to p2p since 2004.
- Scheme since 2013.
- Since 2017 Java development on a 20 year old codebase at Disy Informationssysteme GmbH.
  I now know the other side. :-)
- First lecture on networking in 2018.
- A wonderful wife, two curious kids,
  two guitars, some websites, and a roleplaying game.
  Fighting for time. :-)

What is wisp?

The vision of wisp:

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

Scheme to wisp:

- indentation for outer parentheses
- inline parentheses
- infix math via SRFI-105
- survive HTML (optional)

Why wisp? - Scheme is great!

close to prose

`.,":'_#?!;()`

the most common non-letter non-math characters

flexible

reprogram the compiler for your task

But . . .

But . . .

```
(parens (obscure the)
        (first and last)
        letter)
```

```
parens : obscure the
         first and last
         . letter
```

(and new users shy away from them)

Why wisp? - Elegance

- Elegance 0: generality and homoiconicity (**code is data**).
- Elegance 1: Scheme syntax uses the **most common** non-letter, non-math letters.
- Elegance 2: The **first and last letters** are important for text-recognition.

In teaching, readability is key.

Summary: Why wisp?

Merging the simplicity and elegance of Scheme
with the readability of Python
by reducing parens.

Scheme to wisp: Scheme

```
(define (factorial n)
    "3! = 3 × 2 × 1 = 6"
    (cond
     ((zero? n)
      1)
     (else
      (* n (factorial (- n 1))))))
```

Scheme to wisp: indentation for outer parentheses

```
define (factorial n)
    . "3! = 3 × 2 × 1 = 6"
    cond
      (zero? n)
       . 1
      else
       * n (factorial (- n 1))
```

This is already valid wisp.

Scheme to wisp: inline parentheses

```
define : factorial n
    . "3! = 3 × 2 × 1 = 6"
    cond
       : zero? n
         . 1
       else
        * n : factorial (- n 1)
```

A colon as the **only** element on a line starts a new block:

```
import : srfi srfi-11
let-values
    :
       : x y
         values 1 2
       : z f
         values 3 4
    + x y z f
```

This generalizes wisp to arbitrary tree structures.

Scheme to wisp: infix math with SRFI-105

```
define : factorial n
    . "3! = 3 × 2 × 1 = 6"
    cond
       : zero? n
         . 1
       else
        * n : factorial {n - 1}
```

Main gripe of many.
Use in Scheme:

```
#!curly-infix {1 + 2}
(+ 3 {4 * 5})
```

Scheme to wisp: survive HTML (optional)

```
define : factorial n
_    . "3! = 3 × 2 × 1 = 6"
_    cond
__       : zero? n
___       . 1
__       else
___       * n : factorial {n - 1}
```

Also useful if your LaTeXminted code blocks kill indentation at 8 or more spaces.

Summary: What is wisp?

- **indentation** for outer parentheses
- **leading period** for *"not a procedure call"* (do not prefix the line with a parenthesis)
- **colon** for double parentheses reused for inline parentheses (till the end of the line)

- **infix math** using SRFI-105
- optional leading **underscores** for HTML
- Specified in SRFI-119

5 years with wisp

- 9000 lines of code, pet projects, some in use
- Changes to the language since SRFI-119 (2015-06-23)
  - literal arrays for Guile doctests with ##
  - trailing period for the REPL

→ **wisp as a language is complete and stable.**

9000 lines of code, chronological selection:

- py2guile:* all my **Python** workflows in Guile → Guile basics
- d20world.w: simple **advection and diffusion** on icosahedron
- ensemble-estimation.w:* **kalman filter** function optimization
- enter-three-witches.w: **game scripting** — *thank you cwebber!*
- letterblock-passwords:* `nVxK=8eUD.DdTG`
- network.w: Freenet **network simulator**
- hamming.w:* **error correction**
- downloadmesh.w:* **swarming downloads**, Gnutella style
- fetchpull.w:* multithreaded Freenet **client protocol** library
- dryads-wake.w: **game scripting**

change 1: test-driven wisp: literal arrays for doctests

```
{{{hashbang-and-imports}}}

define : factorial n
    . "3! = 3 × 2 × 1 = 6"
    ## : tests : test-equal 6 : factorial 3
    if : zero? n
       . 1
       * n : factorial {n - 1}

define %this-module : current-module
define : main args
       doctests-testmod %this-module
```

```
%%%% Starting test ._-factorial--factorial
     (Writing full log to "._-factorial--factorial.log")
# of expected passes      1
```

change 2: REPL with wisp: trailing period

```
display "Hello oneliner!\n" .
```

(syntax reserved in SRFI-119 to allow for experimentation)

dryads wake: beginnings of a game

```
define : first-encounter
    Enter : Juli Fin :profile juli
            Rooted Breeze :profile dryad

    Juli Fin
        Finally we have our own home!
    ;; ...
    Rooted Breeze :eerie
        who are you strangers
        in my home?

    Choose
        : explain
          ,(explain-your-home)
        : fast-talk
          ,(fast-talk-the-dryad)
```

dryadswake.webm

Experience with wisp in a lecture

- communication and network technology at DHBW Karlsruhe
- wisp to describe Hamming 11/7 encoding and decoding
- **“Is that pseudocode?”** — a student
  $\rightarrow$ highest praise :-)
- provided as formulary in the (handwritten) final test

Hamming decoder

```
define : 11/7-decode bits
  define broken-bit
    match bits
     : c1 c2 d3 c4 d5 d6 d7 c8 d9 d10 d11
       +
_        * 1 : H c1 d3 d5 d7 d9 d11
_        * 2 : H c2 d3 d6 d7 d10 d11
_        * 4 : H c4 d5 d6 d7
_        * 8 : H c8 d9 d10 d11
  define fixed
    df : zero? broken-bit
       . bits
       flip bits {broken-bit - 1}
  match fixed
    : c1 c2 d3 c4 d5 d6 d7 c8 d9 d10 d11
      list d3 d5 d6 d7 d9 d10 d11
```

Hamming encoder

Header

```
define : 11/7-encode bits
  ##
   tests
    test-equal
     . '(0 0 1 0 0 0 0 1 0 0 1)
     11/7-encode
       . '(1 0 0 0 0 0 1)
```

Body

```
match bits
 : d3 d5 d6 d7 d9 d10 d11
   list
     H d3 d5 d7 d9 d11  ;; bit 1
     H d3 d6 d7 d10 d11 ;; bit 2
     . d3               ;; bit 3
     H d5 d6 d7         ;; bit 4
     . d5 d6 d7         ;; bit 5, 6, 7
     H d9 d10 d11       ;; bit 8
     . d9 d10 d11       ;; bit 9, 10, 11
```

Hamming support procs

```
define : mod2sum . bits
    . "Modulo-2 sum, i.e. for even parity"
    ## : tests : test-equal 1 : mod2sum 1 0 1 1 0
    modulo (apply + bits) 2

define H mod2sum ;; for brevity

define : flip bits index
    . "flip the bit-number (0→1 or 1→0) at the index."
    ## : tests : test-equal '(1 0 1) : flip '(0 0 1) 0
    append
       take bits index
       list : mod2sum 1 : list-ref bits index
       drop bits {index + 1}
```

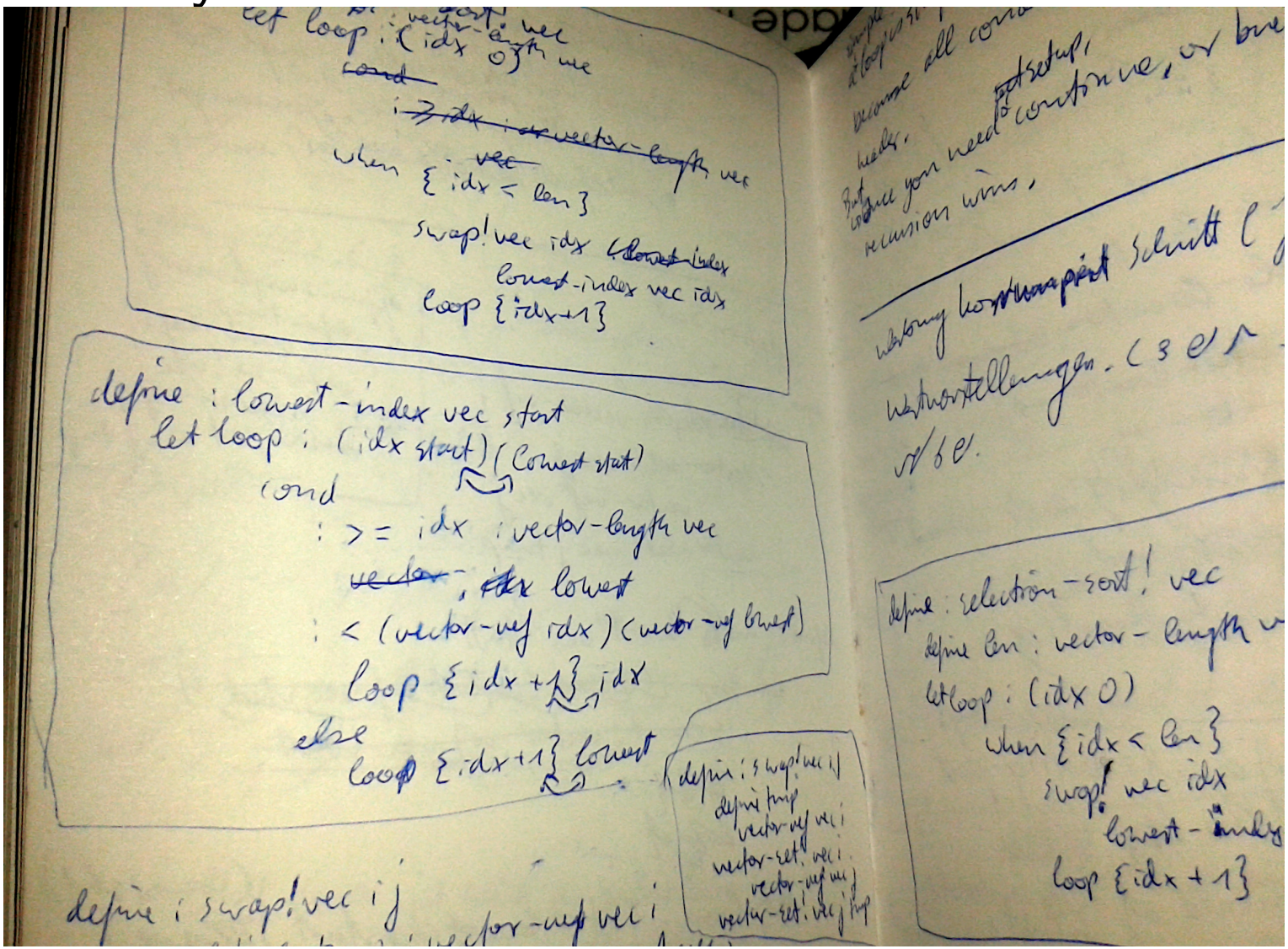
Summary

- **“Is that pseudocode?”**
- Describe calculation in code
- `match` is great for specific examples

Learning: how Scheme and wisp help

- Write code by hand
- Recursion wins: elegance
- Exact math
- Unicode for math

Write code by hand



Recursion wins: elegance

```
define : fib n
   let rek : (i 0) (u 1) (v 1)
         if {i >= {n - 2}}
            . v
            rek {i + 1} v {u + v}
```

Initialize, define parameters, return the result.

Exact math

```
define : n/k n k
   if {k > n} 0
      / : factorial n
          factorial k
          factorial {n - k}
```

No need to work around limitations.

Unicode for math

$$F = \frac{\phi_1 + \phi_2}{2}, G = \frac{\phi_1 - \phi_2}{2}, \lambda = \frac{L_1 - L_2}{2} \tag{1}$$

```
define : ellipsoiddistance a f L1 L2 Φ1 Φ2
    let
        : F : / {Φ1 + Φ2} 2
          G : / {Φ1 - Φ2} 2
          λ : / {L1 - L2} 2
    ;; ...
```

Minimize mental overhead due to mismatch. Math is complex.

Summary

A minimum in the mismatch between task and code.

Best practices I found

- use the weakest method that works*
- use parens and braces where they provide advantages*
- inner defines limit nesting
- do use records
- modules as scripts with doctests

use the weakest method that works

- prefer procedures over macros
- prefer macros over reader extensions

Wisp is the minimal reader extension which can represent arbitrary trees structures with indentation.

use parens where they provide advantages

```
define x^b-deviations-approx
       list-ec (: i ensemble-member-count)
            list-ec (: j (length x^b))
                * : random:normal
                    sqrt : list-ref (list-ref P j) j
```

also:

- use parens for trivial let
- use braces for simple math

Future of wisp (plans and wishes)

- Explore possibilities (as in dryads-wake)
- More documentation (i.e. in With Guise and Guile)
- Better tooling (wisp-mode with paredit commands?)
- Bundle programs cross-platform?
- Part of Guile?

Wisp for pseudocode

The next time you write pseudocode,
try making it executable as wisp

Wisp for pseudocode

The next time you write pseudocode,
try making it executable as wisp

. . . and talk about it!

»ArneBab's alternate sexp syntax is best I've seen; pythonesque, hides parens but keeps power« — Christopher Lemmer Webber in Wisp: Lisp, minus the parentheses

Try wisp!

Install

```
guix package -i guile guile-wisp
```

REPL

```
guile -L . -x .w --lanugage=wisp
```

More info

https://www.draketo.de/english/wisp

Wisp for scripts with guix

```
#!/run/current-system/profile/bin/bash
# -*- wisp -*-
exec -a "$0" guile -L $(dirname $(realpath "$0")) \
     -x .w --language=wisp -e '(script)' -c '' "$@"
; !#

define-module : script
   . #:export : main

define : main args
    format #t "Hello Wisp!"
```

Wisp resources

- Website: https://www.draketo.de/english/wisp
- Tutorial: https://www.draketo.de/proj/with-guise-and-guile/wisp-tutorial.html
- Examples: https://bitbucket.org/ArneBab/wisp/src/tip/examples
- guile-freenet: https://notabug.org/arnebab/guile-freenet
- dryads wake: https://bitbucket.org/ArneBab/dryads-wake

Thank you for listening!

nonlocal state

```
(opened (parens)
  (are (nonlocal
  state)))
```

```
opened : parens
  are
    nonlocal state
```

(you or your tooling must track them)

Wisp for scripts anywhere

```
#!/usr/bin/env bash
# -*- wisp -*-
D=$(dirname $(realpath "$0"))
# precompile wisp
guile -L "$D" -c '(import (language wisp spec))'
# run script as wisp code
exec -a "$0" guile -L "$D" \
     -x .w --language=wisp -e '(script)' -c '' "$@"
; !#

define-module : script
    . #:export : main

define : main args
    format #t "Hello Wisp!"
```

Verweise I

Bilder: