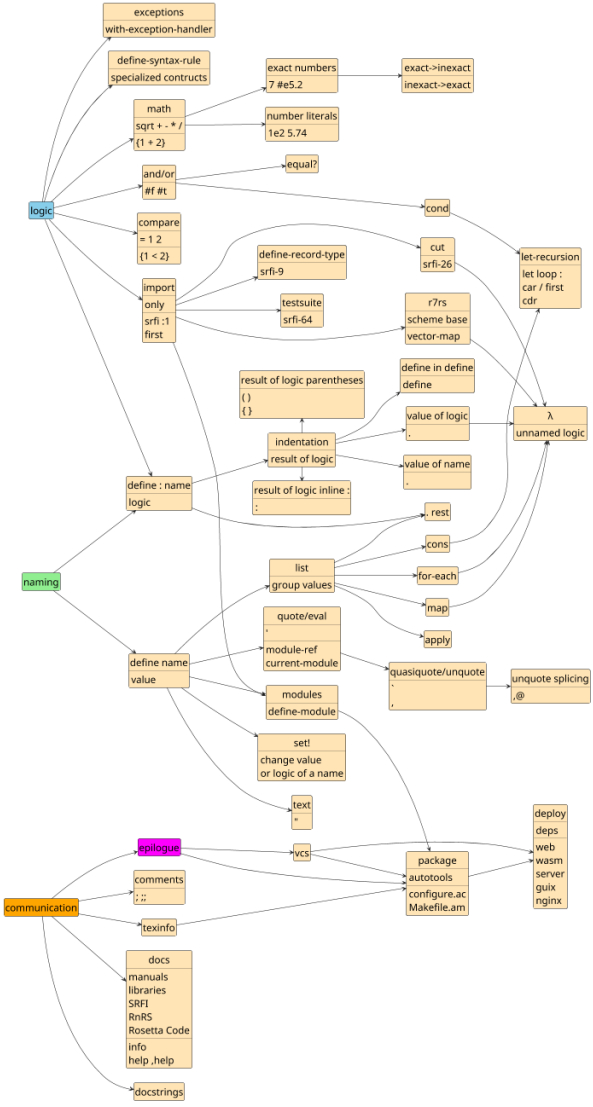
Dr. Arne Babenhauserheide / draketo.de

Naming and Logic (Sche

and a three-fold Zen for Scheme

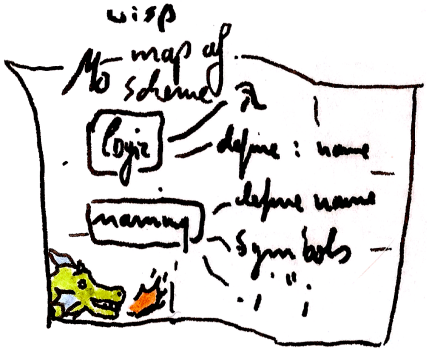
programming essentials with Scheme



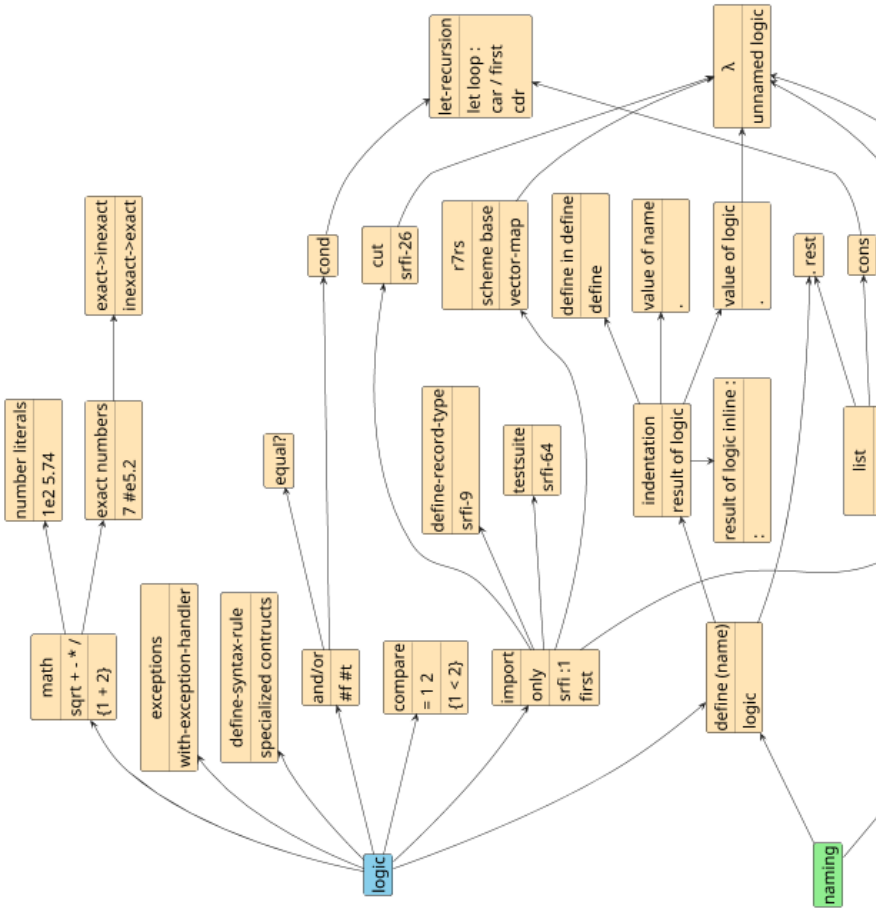
Find the heart of programming with the map of Scheme.

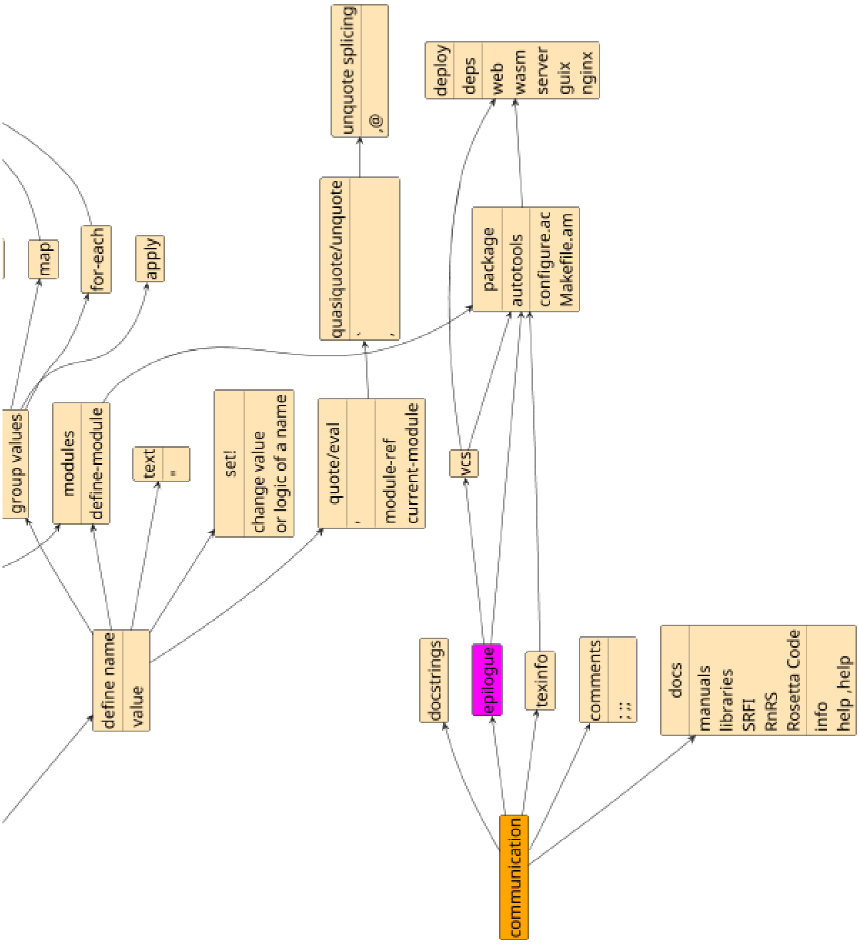
To follow along, install Guile and try the examples as you read.

The extension language of the GNU project.



i The Map of Scheme




group values
map
for-each
apply
modules
define-module
text
"
set!
change value
or logic of a name
define name
value
quote/eval
'
module-ref
current-module
quasiquote/unquote
`
,
unquote splicing
,@
deploy
deps
web
wasm
server
guix
nginx
package
autotools
configure.ac
Makefile.am
vcs
docstrings
epilogue
texinfo
comments
;;;
docs
manuals
libraries
SRFI
RnRS
Rosetta Code
info
help ,help
communication

Contents

ii Preface

Why this book? Providing a concise start, a no-frills, opinionated intro to programming from first `define` to deploying an application on just 64 short pages.

Who is it for? You are a newcomer and want to **learn by trying code examples**? You know programming and want **a running start** into Scheme? You want to see how little suffices with Scheme's Then this book is for you.

What is Scheme? Scheme is a programming language — a Lisp — that follows principle *"design not by piling feature on top of feature, but by **removing the weaknesses and restrictions** that make additional features appear necessary"*. This book uses Guile Scheme, the official extension language of the GNU project.

How to get Guile? Download and install Guile from the website `www.gnu.org/software/guile` — then open the REPL by executing `guile` in the terminal. The REPL is where you type and try code interactively. On some platforms you need to use `guile3.0`.

Text, Design, and Publishing: Arne Babenhauserheide,
Karlsruher Str. 85,
76676 Graben-Neudorf
arne_bab@web.de

Production: epubli – ein Service der neopubli GmbH,
Köpenicker Straße 154a, 10997 Berlin
Contact according to EU product safety regulation:
produktsicherheit@epubli.com

1 Name a value: define

Use `(define ...)` to name a value. Return the last name.

```
(define small-tree-height-meters 3)
(define large-tree-height-meters 5)
small-tree-height-meters
```

After typing or copying a block into the Guile REPL, **hit enter**. You should then see

```
$1 = 3
```

This means: the first returned value (`$1`) is 3. The next time you return a value, it will be called `$2`.

Names can contain any letter except for (white-)space, quote, comma or parentheses. They must not be numbers.

```
(define illegal name 1)
(define 'illegal-name 2)
(define ,illegal-name 3)
(define illegal)name 4)
(define 1113841 5)
```

```
While executing meta-command:
Syntax error:
unknown location: source expression failed to match any
↪  pattern in form (define illegal name 1)
While reading expression:
#<unknown port>:14:2: unexpected ")"
```

2 Compare numbers

```
(= 3 3)
```

```
#t
```

```
(= 3 5)
```

```
#f
```

`#t` means true, `#f` means false. Parentheses return the result of logic. The logic comes first. This is clear for `=`, but easy to misread for `<`.

```
(< 3 5) ;; is 3 smaller than 5? #true
```

```
#t
```

```
(> 5 3) ;; is 5 bigger than 3? #true
```

```
#t
```

```
(> 3 3) ;; is 3 bigger than 3? #false
```

```
#f
```

```
(>= 3 3) ;; is 3 bigger or equal to 3? #true
```

```
#t
```

```
(<= 3 3) ;; is 3 bigger or equal to 3? #true
```

```
#t
```

3 Use infix in logic

```
#!curly-infix
{3 = 5}
```

```
#f
```

```
#!curly-infix
{3 < 5}
```

```
#t
```

Infix logic directly gives a value. To use it, you must put `#!curly-infix` somewhere in your code before you use the first curly brace (`{...}`).

Because infix-logic gives a value, you can use it in place of a value, for example to nest or name it:

```
#!curly-infix
{{5 < 3} equal? #f}
```

```
#t
```

```
#!curly-infix
(define is-math-sane? {3 < 5})
is-math-sane?
```

```
#t
```

By convention, names that have the value `true` or `false` have the suffix `?`.

4 Use named values in logic

```
#!curly-infix
(define small-tree-height/m 3)
(define large-tree-height/m 5)
{small-tree-height/m < large-tree-height/m}
```

```
#t
```

5 Add comments with ;

```
(define birch-height/m 3)
;; this is a comment
(define height ;; comment at the end
  ;; comment between lines
  birch-height/m)
height
```

```
3
```

It is common to use ;; instead of ;, but not required.

6 Logic with true and false

```
(and #t #t)
```

```
#t
```

```
(and #f #t)
```

```
#f
```

```
(or #f #t)
```

```
#t
```

```
(or #f #f)
```

```
#f
```

If any value passed to `and` is `#f` (`#false`), it ignores further values. If any value passed to `or` is not `#f` (not `#false`), it ignores further values.

```
#!curly-infix
(and #t #t #t) ;; => #true
(and #t #f #t) ;; => #false
(and {3 < 5} {5 < 3}) ;; => #false
(or #t #f #t) ;; => true
(or {3 < 5} {5 < 3}) ;; => #true
(or #f #f #f) ;; => #false
```

For `and` and `or`, everything is `#true` (`#t`) except for `#false` (`#f`). Given the number of hard to trace errors in other languages that turn up in production, this is the only sane policy.

7 Name the result of logic with indentation

```
#!curly-infix
(define birch-h/m 3)
(define chestnut-h/m 5)
(define same-héight?
  (= birch-h/m chestnut-h/m))
(define smaller?
  {birch-h/m < chestnut-h/m}) ;; infix
smaller?
```

```
#t
```

The infix gives a value, so it directly returns its value.

8 Name logic with `define (`

```
(define (same-height? tree-height-a tree-height-b)
  (= tree-height-a tree-height-b))
(same-height? 3 3)
```

```
#t
```

By convention, logic that returns true or false has the suffix `?`.

You can now use your named logic like all other logic. Even with infix.

```
#!curly-infix
(define (same-height? tree-height-a tree-height-b)
  (= tree-height-a tree-height-b))
{3 same-height? 3}
```

```
#t
```

> What this map of Scheme calls *named logic* is commonly called `function` or `procedure`. We'll stick with *logic* for the sake of a leaner conceptual mapping.

The indented lines with the logic named here are called the **body**.

9 Name a name with `define`

```
(define small-tree-height-meters 3)
(define height
  small-tree-height-meters)
height
```

```
3
```

10 Return the value of logic

```
#!curly-infix
(define (larger-than-4? size)
  {size > 4})
larger-than-4?
```

```
#<procedure larger-than-4? (size)>
```

The value of logic defined with `define` `(` is a `procedure`. You can see the arguments in the output: If you call it with too few or too many arguments, you get warnings.

There are other kinds of logic: syntax rules and reader-macros. We will cover syntax rules later. New reader macros are rarely needed; using `{...}` for infix math is a reader macro activated with `#!curly-infix`.

11 Name in `define (` with `define`

```
#!curly-infix
(define birch-h/m 3)
(define (birch-is-small)
  (define reference-h/m 4)
  {birch-h/m < reference-h/m})
(birch-is-small)
```

```
#t
```

Only the last part of the body of `define (` is returned.

A calculation inside curly braces is executed in-place, so when it is the last element, its result value is returned.

A Zen for Scheme part 1: Birds Eye

RR Remove limitations to Reduce the feature-count you need,
*but **OM**: Optimizability Matters.*
FI Freedom for the Implementations and from Implementations,
*but **CM**: Community Matters: Join the one you choose.*
SL Mind the Small systems!
And the Large systems!
ES Errors should never pass silently,
unless speed is set higher than safety.

Thanks for the error-handling principle goes to John Cowan.

12 Return list of values with `list`

```
(define known-heights
  (list 3 3.75 5 100))
(list (list 3 5)
      known-heights)
```

```
((3 5) (3 3.75 5 100))
```

You can put values on their own lines. Different from `define (`, list keeps all values, not just the last.

```
(define known-heights-2
  (list 3
     3.75 5
     100)) ;; ist keeps all numbers
(define known-heights-3
  (list
    3
    3.75
    5
    100))
(define (last-height)
  3 3.75 5 100) ;; (define () ...) only keeps 100
(= 100 (last-height))
```

```
#t
```

13 Name the result of logic in one line with ()

```
(define birch-h/m 3)
(define chestnut-h/m 5)

(define same-height (= birch-h/m chestnut-h/m))
same-height
```

```
#f
```

This is consistent with infix-math and uniform with defining logic:

```
#!curly-infix
(define birch-h/m 3)
(define chestnut-h/m 5)

(define same-height {birch-h/m = chestnut-h/m})

(define (same-height? tree-height-a tree-height-b)
  (= tree-height-a tree-height-b))
(define same-height2 (same-height? birch-h/m
↪  chestnut-h/m))
(list same-height same-height2)
```

```
'(#f #f)
```

14 Name text with "

```
(define tree-description "large tree")
(define footer "In Love,

Arne")
(define greeting
  "Hello")
(display footer)
```

```
In Love,

Arne
```

Like `{ }`, text (called `string` as in "string of characters") is its value.

Text can span multiple lines. Line breaks in text do not affect the meaning of code.

You can use `\n` to add a line break within text without having a visual line break. The backslash (\) is the escape character and `\n` represents a line break. To type a real \ within quotes ("), you must escape it as \\.

With `display` you can show text as it will look in an editor.

Text is stronger than comments:

```
(define with-comment ;; belongs to coment
  ;; comment
  "Hello ;; part of the text")
with-comment
```

```
Hello ;; part of the text
```

15 Take decisions with `cond`

```
#!curly-infix
(define chestnut-h/m 5)
(define tree-description
  (cond
    ({chestnut-h/m > 4}
      "large tree")
    ((= 4 chestnut-h/m)
      "four meter tree")
    (else
      "small tree")))
tree-description
```

```
large tree
```

`cond` checks its clauses one by one and uses the first with *value* `#true`.
To `cond` every valid value is `#true` (`#t`) except for `#false` (`#f`).
To use named logic, prefix it with `:` to check its *value*.

```
#!curly-infix
(list
 (cond
  (5 #t)
  (else ;; else is #true in cond
   #f))
 (cond (#f #f)
       (else #t))
 (cond
  ({3 < 5} #t)
  (else #f)))
```

```
'(#t #t #t)
```

16 Use fine-grained numbers with number-literals

```
(define more-precise-height 5.32517)
(define 100-meters 1e2)
(list more-precise-height
      100-meters)
```

```
(5.32517 100.0)
```

These are floating point numbers. They store approximate values in 64 bit binary, depending on the platform. Read all the details in the Guile Reference manual Real and Rational Numbers, the r5rs numbers, and IEEE 754.[1]

17 Use exact numbers with `#e` and quotients

```
(define exactly-1/5 #e0.2)
(define exactly-1/5-too 1/5)
(list exactly-1/5
      exactly-1/5-too)
```

```
(1/5 1/5)
```

Guile computations with exact numbers stay reasonably fast even for unreasonably large or small numbers.

[1] All links are listed on page 64.

18 See inexact value of exact number with `exact->inexact`

```
(list (exact->inexact #e0.2)
      (exact->inexact 1/5)
      (exact->inexact 2e7))
```

```
(0.2 0.2 2.0e7)
```

The inverse is `inexact->exact`:

```
(inexact->exact 0.5)
```

```
1/2
```

Note that a regular `0.2` need not be exactly `1/5`, because floating point numbers do not have exact representation for that. You'll need `#e` to have precise `0.2`.

```
(list (inexact->exact 0.2)
      #e0.2)
```

```
(3602879701896397/18014398509481984 1/5)
```

19 Use math with the usual operators as logic

```
(define one-hundred
  (* 10 10))
(define half-hundred (/ one-hundred 2))
half-hundred
```

```
50
```

Remember that names cannot be valid numbers!

```
(define 100 ;; error!
  (* 10 10))
```

```
ice-9/boot-9.scm:1685:16: In procedure raise-exception:
Syntax error:
unknown location: source expression failed to match any
↪  pattern in form (define 100 (* 10 10))

Entering a new prompt.  Type `,bt' for a backtrace or
↪  `,q' to continue.
```

20 Compare structural values with equal?

```
;; reuse name definition snippets from
;; Return list of values with =list=
{{{known-heights}}}
{{{known-heights2}}}
(equal? known-heights known-heights-2 known-heights-3)
```

```
#t
```

Like = and +, `equal?` can be used on arbitrary numbers of values.

Reusing the snippets from Return list of values with `list` uses `noweb` syntax via Emacs Org Mode, the .

A Zen for Scheme part 2: On the Ground

HA Hygiene reduces Anxiety,
except where it blocks your path.
PP Practicality beats Purity,
except where it leads into a dead end.
3P 3 Pillars of improvement:
Experimentation, Implementation, Standardization.

21 Apply logic to a list of values with `apply`

```
(apply = (list 3 3))
```

```
#t
```

```
(equal?
  (= 3 3)
  (apply =
    (list 3 3)))
```

```
#t
```

Only the last argument of apply is treated as list, so you can give initial arguments:

```
(define a 1)
(define b 1)
(apply = a b
  (list 1 1)) ;; becomes (= a b 1 1)
```

```
#t
```

22 Get the arguments of named logic as list with . args

```
(define (same? heights)
  (apply = heights))
(define (same2? . heights)
  (apply = heights))
(list (same? (list 1 1 1))
      (same2? 1 1 1))
```

```
'(#t #t)
```

These are called `rest`. Getting them is not for efficiency: the list creation is implicit. You can mix regular arguments and `rest` arguments:

```
(define (same? alice bob . rest)
  (display (list alice bob rest))
  (newline)
  (apply = alice bob rest))
(display (same? 1 1 1 1))
```

```
(1 1 (1 1))
#t
```

Remember that apply uses only the last of its arguments as list, in symmetry with . `rest`.

23 Change the value or logic of a defined name with `set!`

```
(define birch-h/m 3)
(set! birch-h/m 3.74)
birch-h/m
```

```
3.74
```

It is customary to suffix named logic that changes values of existing names with `!`.

Since logic can cause changes to names and not just return a result, it is not called `function`, but `procedure` in documentation; `proc` for brevity.

24 Apply logic to each value in lists and ignoring the results with `for-each`

```
#!curly-infix
(define birch-h/m 3)
(define has-birch-height #f)
(define (set-true-if-birch-height! height/m)
  (cond
    ({birch-h/m = height/m}
      (set! has-birch-height #t))))
(define heights (list 3 3.75 5 100))
(for-each set-true-if-birch-height! heights)
has-birch-height
```

```
#t
```

25 Get the result of applying logic to each value in lists with `map`

```
(define birch-h/m 3)
(define (same-height-as-birch? height/m)
  (= birch-h/m height/m))
(define heights (list 3 3.75 5 100))
(list heights
      (map same-height-as-birch?
        heights)
      (map + ;; becomes 1+3 2+2 3+1
        (list 1 2 3)
        (list 3 2 1))
      (map list
        (list 1 2 3)
        (list 3 2 1)))
```

```
'((3 3.75 5 100) (#t #f #f #f) (4 4 4) ((1 3) (2 2) (3 1)))
```

When operating on multiple lists, `map` takes one argument from each list. All lists must be the same length. *To remember*: `apply` extracts the values from its *last argument*, `map` extracts one value from *each argument* after the first. `apply map list ...` flips colums and rows:

```
(apply map list
  (list (list 1 2 3)
        (list 3 2 1)))
```

```
((1 3) (2 2) (3 1))
```

26 Create nameless logic with `lambda`

```
(define (is-same-height? a b)
  (> a b)) ;; <- this is a mistake
(display is-same-height?)(newline)
(display (is-same-height? 3 3))(newline)
(define (fixed a b)
  (= a b))
(set! is-same-height? fixed)
(display is-same-height?) ;; now called "fixed"
(newline)
(display (is-same-height? 3 3))(newline)
;; shorter and avoiding name pollution and confusion.
(set! is-same-height?
  (lambda (a b)
    (= a b))) ;; must be on a new line
          ;; to not be part of the arguments.
;; since lambda has no name, we see the original name
(display is-same-height?)
(display (is-same-height? 3 3))
```

```
#<procedure is-same-height? (a b)>
#f
#<procedure fixed (a b)>
#t
#<procedure is-same-height? (a b)>#t
```

The return value of `lambda` is logic (a `procedure`).

If logic is defined via `define (`, it knows the name it has been defined as. With `lambda`, it does not know the name.

`lambda` is a special form. Think of it as
`(define (name arguments) ...)`, but without the name.

27 Reuse your logic with `let-recursion`

Remember the `for-each` example:

```
#!curly-infix
(define has-birch-height #f)
(define heights (list 3 3.75 5 100))
(define (set-true-if-birch-height! height/m)
  (define birch-h/m 3)
  (cond
    ({birch-h/m = height/m}
      (set! has-birch-height #t))))
(for-each set-true-if-birch-height! heights)
has-birch-height
```

```
#t
```

Instead of `for-each`, we can build our own iteration:

```
(define heights (list 3 3.75 5 100))
(define (has-birch-height? heights)
  (define birch-h/m 3)
  (let loop ((heights heights))
    (cond
      ((null? heights) #f)
      ((= birch-h/m (car heights)) ;; car is first
        #t)
      (else
        (loop (cdr heights))))))
(has-birch-height? heights)
```

```
#t
```

`null?` asks whether the list is empty. `car` gets the first element of a list, `cdr` gets the list without its first element.

Recursion is usually easier to debug (all variable elements are available at the top of the let recursion) and often creates cleaner APIs than iteration.

As rule of thumb: start with the recursion end condition (here: `(null? heights)` and ensure that each branch of the `cond` either ends recursion or moves a step towards finishing (usually with `cdr`).

Another example why recursion wins:

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

Zen for Scheme

A Zen for Scheme part 3: Submerged in Code

WM Use the Weakest Method that gets the job done,
but know the stronger methods to employ them as needed.

RW Recursion Wins,
except where a loop-macro is better.

RM Readability matters,
and nesting works.

28 Import pre-defined named logic and values with `import`

```
(import (ice-9 pretty-print)
        (srfi :1 lists))

(pretty-print
  (list 12
    (list 34)
    5 6))
(pretty-print (list
  (first (list 1 2 3)) ;; 1
  (second (list 1 2 3)) ;; 2
  (third (list 1 2 3)) ;; 3

  (member 2 (list 1 2 3)))) ;; list 2 3 => #true
```

```
(12 (34) 5 6)
(1 2 3 (2 3))
```

Import uses modules which can have multiple components. In the first import, `ice-9` is one component and the second is `pretty-print`. In the second, `srfi` is the first component, `:1` is the second, and `lists` is the third.

ice-9 is the name for the core extensions of Guile. It's a play on ice-9, a fictional perfect seed crystal.

SRFI's are Scheme Requests For Implementation, portable libraries built in collaboration between different Scheme implementations. The ones available in Guile can be found in the Guile Reference manual. More can be found on srfi.schemers.org. They are imported by number (`:1`) and can have a third component with a name, but that's not required.

You can use `only` to import only specific names.

```
(import (only (srfi :1) first second) ;; no third
        (only (srfi :1) iota))

(first (list 1 2 3)) ;; 1
(second (list 1 2 3)) ;; 2
(third (list 1 2 3)) ;; error
(iota 5) ;; list 0 1 2 3 4
```

```
ice-9/boot-9.scm:1685:16: In procedure raise-exception:
Unbound variable: third

Entering a new prompt.  Type `,bt' for a backtrace or
↪  `,q' to continue.
```

29 Extend a list with `cons`

The core of composing elementwise operations.

To build your own `map` function which returns a list of results, you need to add to a list.

```
(cons 1 (list 2 3))
;; => list 1 2 3
```

1 2 3

Used for a simplified `map` implementation taking a single list:

```
(import (only (srfi :1) first))
(define (single-map proc elements)
  (let loop ((changed (list)) (elements elements))
    (cond
      ((null? elements)
        (reverse changed))
      (else
        (loop
          ;; add processed first element to changed
          (cons (proc (first elements))
                changed)
          ;; drop first element from elements
          (cdr elements))))))
(single-map even? (list 1 2 3))
;; => #f #t #f
```

```
'(#f #t #f)
```

30 Apply partial procedures with `srfi :26 cut`

```
(import (srfi :26 cut))

(define (plus-3 number)
  (+ 3 number))
(define plus-3-cut (cut + 3 <>))

(list
  (map plus-3
    (list 1 2 3)) ;; list 4 5 6

  (map (cut + 3 <>)
    (list 1 2 3)) ;; list 4 5 6

  (map (cut - <> 1) ;; => <> - 1
    (list 1 2 3)) ;; list 0 1 2

  (map (cut - 1 <>) ;; => 1 - <>
    (list 1 2 3)) ;; list 0 -1 -2

  (map plus-3-cut
    (list 1 2 3))) ;; list 4 5 6
```

```
((4 5 6) (4 5 6) (0 1 2) (0 -1 -2) (4 5 6))
```

31 Use `r7rs` datatypes, e.g. with `vector-map`

R^{7}RS is the 7th Revised Report on Scheme. Guile provides a superset of the standard: its core can be imported as `scheme base`. A foundational datatype is Vectors with `O(1)` random access guarantee.

```
(import (scheme base))
(define vec (list->vector '(1 b "third")))
(vector-map (λ (element) (cons 'el element))
            vec)
```

```
#((el . 1) (el . b) (el . "third"))
```

Vectors have the literal form `#(a b c)`. It is an error to mutate these.

```
(import (scheme base))
(define mutable-vector (list->vector '(1 b "third")))
(define literal-vector #(1 b "third"))
(vector-set! mutable-vector 1 "bee") ;; allowed
(vector-set! literal-vector 1 "bee") ;; forbidden
(list mutable-vector literal-vector)
```

32 Name structured values with `define-record-type`

```
(import (srfi :9 records))

(define-record-type <tree>
  (make-tree type height-m weight-kg carbon-kg)
  tree?
  (type tree-type)
  (height-m tree-height)
  (weight-kg tree-weight)
  (carbon-kg tree-carbon))

(define birch-young
  (make-tree "birch" 13 90 45)) ;; 10 year, 10cm diameter,
(define birch-old
  (make-tree "birch" 30 5301 2650)) ;; 50 year, 50cm
(define birch-weights
  (map tree-weight (list birch-young birch-old)))
(list birch-young
      birch-old
      birch-weights)
```

```
'(#<<tree> type: "birch" height-m: 13 weight-kg: 90
↪  carbon-kg: 45> #<<tree> type: "birch" height-m: 30
↪  weight-kg: 5301 carbon-kg: 2650> (90 5301))
```

Carbon content in birch trees is about 46% to 50.6% of the mass. See forestry commission technical paper 1993.

Height from Waldwissen, weight from BaumUndErde.

33 Create your own modules with `define-module`

To provide your own module, create a file named by the module name. For `(import (example trees))` the file must be `example/trees.scm`. Use `define-module` and `#:export` what gets imported:

```
(define-module (example trees)
               #:export
               (birch-young
                 make-tree tree? tree-carbon))
(import (srfi :9 records)) ;; imports after module
(define-record-type <tree> ;; reduced
  (make-tree carbon-kg)
  tree?
  (carbon-kg tree-carbon))
(define birch-young
  (make-tree 45)) ;; 10 year, 10cm diameter,
```

To use that module, add your root folder to the search path. Then just import it. To ensure that the file is run that way, use shell-indirection:

```
#!/usr/bin/env bash
exec -a "${0}" guile \
     -L "$(dirname "${0}")" -x .w $@
;; !# Guile execution
(import (example trees))
birch-young
```

```
$1 = #<<tree> carbon-kg: 45>
```

Make executable with `chmod +x the-file.w`, run with `./the-file.w`

34 Handle errors `with-exception-handler`

```
;; unhandled exception stops execution
(define (add-5 input)
  (+ 5 input)) ;; illegal for text
(map add-5 '("five" 6 "seven"))
;; check inputs
(define (add-5-if input)
  (if (number? input)
      (+ 5 input)
      #f))
(map add-5-if '("five" 6 "seven"))
```

```
ice-9/boot-9.scm:1685:16: In procedure raise-exception:
In procedure +: Wrong type argument in position 1: "five"

Entering a new prompt.  Type `,bt' for a backtrace or `,q' to conti
```

```
;; handle exceptions
(define (add-5-handler input)
  (with-exception-handler
    (λ (e) (format #t "must be number, is ~S.\n" input)
           #f)
    (λ () (+ 5 input))
    #:unwind? #t)) ;; #t: continue #f: stop
(map add-5-handler '("five" 6 "seven"))
```

```
must be number, is "five".
must be number, is "seven".
```

In Guile Scheme checking inputs is often cheaper than exception handling.

35 Test your code with `srfi 64`

Is your code correct?

```
(import (srfi :64 testsuite))

(define (tree-carbon weight-kg)
  (* 0.5 weight-kg))

(define (run-tests)
  (test-begin "test-tree-carbon")
  (test-equal 45.0
    (tree-carbon 90))
  (test-approximate 45.0
    (+ 40 (random 10.0))
    5) ;; expected error size
  (test-assert (equal? 45.0 (tree-carbon 90)))
  (test-error (throw 'wrong-value))
  (test-end "test-tree-carbon"))

(run-tests)
```

```
%%%% Starting test test-tree-carbon  (Writing full log to "test-tre
# of expected passes      4
```

You can use this anywhere.

For details, see srfi 64.

36 Define derived logic structures with `define-syntax-rule`

In usual logic application in `procedures`, arguments are evaluated to their return value first. `Procedures` evaluate from **inside to outside**:

```
(import (ice-9 pretty-print))
(define (hello-printer . args)
  (pretty-print "Hello")
  (for-each pretty-print args))
(hello-printer 1 (pretty-print "second")
               3 4)
;; prints "second" "Hello" 1 3 4
```

```
"second"
"Hello"
1
#<unspecified>
3
4
```

But for example `cond` only evaluates the required branches. It is not a `procedure`, but a `syntax-rule`. Syntax-rules evaluate from **outside to inside**:

```
(import (ice-9 pretty-print))
(define-syntax-rule (early-printer args ...)
  (begin
    (pretty-print "Hello")
    (for-each pretty-print (list args ...))))
(early-printer 1 (pretty-print "second")
               3 4)
;; prints "Hello" "second" 1 3 4
```

```
"Hello"
"second"
1
#<unspecified>
3
4
```

Arguments of `define-syntax-rule` are only evaluated when they are passed into a regular `procedure` or returned. By calling other `syntax-rules` in `syntax-rules`, evaluation can be delayed further.

`define-syntax-rule` can reorder arguments and pass them to other `syntax-rules` and to `procedures`. It cannot ask for argument values, because it does not evaluate names as values. It operates on names and structure.

Instead of `(define (name . args) ...)`, it uses a pattern:

`(define-syntax-rule (name args ...) ...)`

The ellipsis `...` marks `args` as standing for zero or more names. It must be used with the ellipsis.

The body of `define-syntax-rule` must only have one element. The logic `begin` wraps its own body to count as only one element. It returns the value of the last element in its body.

37 Get and resolve names used in code with `quote`, `eval`, and `module-ref`

```
(list (quote alice)
      (quote bob)
      (quote carol)
      (quote dave))
;; => (alice bob carol dave)

(define alice "the first")

(eval 'alice (current-module))
;; => "the first"
(module-ref (current-module) 'alice)
;; => "the first"
;; module-ref is less powerful than eval. And safer.

(define code
  (quote
    (list 1 2 3)))
code
;; => (list 1 2 3)
;;    uses parentheses form
(eval code (current-module))
;; => (1 2 3)

'(1 2 3)
;; (1 2 3)
(list 1 2 3)
;; (1 2 3)

(equal? '(1 2 3)
      (list 1 2 3))
```

The form `'(1 2 3)` is a shorthand to create an **immutable** (literal) list that is `equal?` to `list 1 2 3`.

But some operations like `list-set! the-list index new-value` from `srfi :1` do not work on immutable lists.

```
(define mutable-list (list 1 2 3))
(list-set! mutable-list 1 'a) ;; zero-indexed
mutable-list
(define immutable-list '(1 2 3))
immutable-list
(list-set! immutable-list 1 'a) ;; error!
```

```
ice-9/boot-9.scm:1685:16: In procedure raise-exception:
In procedure set-car!: Wrong type argument in position 1
↪  (expecting mutable pair): (2 3)

Entering a new prompt.  Type `,bt' for a backtrace or
↪  `,q' to continue.
```

38 Build value-lists with `quasiquote` and `unquote`

```
(define (tree-manual type height weight carbon-content)
  (list (cons 'type type)
        (cons 'height height )
        (cons 'weight weight)
        (cons 'carbon-content carbon-content)))
(tree-manual "birch" 13 90 45)

(define (tree-quasiquote type height weight
                         carbon-content)
  (quasiquote
    ((type . (unquote type))
     (height . (unquote height))
     (weight . (unquote weight))
     (carbon-content . (unquote carbon-content)))))
(tree-quasiquote "birch" 13 90 45)

(define (tree-shorthand type height weight carbon-content)
  `((type . ,type    ) ;; ` is short for quasiquoted list
    (height . ,height) ;; , is short for unquote
    (weight . ,weight)
    (carbon-content . ,carbon-content)))
(tree-shorthand "birch" 13 90 45)
```

These three methods are almost equivalent, except that quasiquoting can create an immutable list, but this is not guaranteed.

```
(define three 3)
(define mutable-list (list 1 2 3))
(list-set! mutable-list 1 'a) ;; zero-indexed
mutable-list
(define immutable-list `(1 2 3))
(list-set! immutable-list 1 'a) ;; error!
immutable-list
(define mutable-quasiquoted `(1 2 ,three))
(list-set! mutable-quasiquoted 1 'a) ;; no error yet!
mutable-quasiquoted
```

```
ice-9/boot-9.scm:1685:16: In procedure raise-exception:
In procedure set-car!: Wrong type argument in position 1
↪  (expecting mutable pair): (2 3)

Entering a new prompt.  Type `,bt' for a backtrace or
↪  `,q' to continue.
```

Mutating quasiquoted lists may throw an error in the future. From the standard:

> A quasiquote expression may return either newly allocated, mutable objects or literal structure for any structure that is constructed at run time ...

39 Merge lists with `append` or `unquote-splicing`

```
(import (ice-9 pretty-print))
(define birch-carbon/kg '(5000 5301 5500))
(define oak-carbon/kg '(7000 7700 8000))
;; append merges lists
(append birch-carbon/kg
  oak-carbon/kg)
;; unquote-splicing splices a list into quasiquote (`)
(pretty-print
  `((unquote-splicing birch-carbon/kg)
    (unquote-splicing oak-carbon/kg)))
;; with shorthand ,@
(pretty-print
  `(,@birch-carbon/kg
    ,@oak-carbon/kg))
```

```
(5000 5301 5500 7000 7700 8000)
(5000 5301 5500 7000 7700 8000)
```

Unquote splicing can also insert the result of logic:

```
`(,@(map 1- '(1 2 3))
  ,@(map 1+ (reverse '(0 1 2)))
  (unquote-splicing (list 0)))
```

```
(0 1 2 3 2 1 0)
```

40 Document procedures with docstrings

```
(define (documented-proc arg)
  "Proc is documented"
  #f) ;; doc must not be last element
(procedure-documentation documented-proc)
;; variables have no docstrings but
;; properties can be set manually.
(define variable #f)
(set-object-property! variable 'documentation
  "Variable is documented")
(object-property variable 'documentation)
```

You can get the documentation with `help` or `,d` on the REPL:

```
,d documented-proc => Proc is documented
,d variable => Variable is documented
```

For generating documentation from comments, there's `guild doc-snarf`.

```
;; Proc docs can be snarfed
(define (snarfed-proc arg)
  #f)
;; Variable docs can be snarfed
(define snarfed-variable #f)
```

If this is saved as `hello.scm`, get the docs via

```
guild doc-snarf --texinfo hello.scm
```

41 Read the docs

Now you understand the heart of code. With this as the core there is one more step, the lifeblood of programming: learning more. Sources:

- the Guile Reference manual
- the Guile Library
- Scheme Requests for Implementation (SRFI): tagged libraries
- The Scheme standards (RnRS), specifically r7rs-small (pdf)
- a list of tools and libraries
- Rosetta Code with solutions to many algorithm problems

Info manuals can often be read online, but the `info` commandline application and `info` in Emacs (`C-h i`) are far more efficient and provide full-text search. You can use it to read the Guile reference manual and some libraries. Get one by installing texinfo or Emacs.

In **interactive** `guile` (the REPL), you can check documentation:

```
(help string-append)
```

```
`string-append' is a procedure in the (srfi srfi-13) module.

- Scheme Procedure: string-append .  args
     Return a newly allocated string whose characters form the
     concatenation of the given strings, ARGS.
```

```
,help
```

```
Help Commands [abbrev]:
...
```

Note: the full links are printed in the list of links on page 64.

42 Create a manual as package documentation with `texinfo`

Create a `doc/` folder and add a `hello.texi` file.

An **example file** can look like the following:

```
@documentencoding UTF-8
@settitle Hello World
@c This is a comment; The Top node is the first page
@node Top

@c Show the title and clickable Chapter-names as menu
@top
@menu
* First Steps::
* API Reference::
@end menu

@contents
@node First Steps
@chapter First Steps
@itemize
@item
Download from ...
@item
Install: @code{make}.
@end itemize

Example:
@lisp
(+ 1 2)
@result{} 3
@end lisp
```

```
@node API Reference
@chapter API Reference
@section Procedures
@subsection hello
Print Hello
@example
hello
@end example
```

Add a `Makefile` in the doc/ folder:

```
all: hello.info hello.epub hello_html/index.html
hello.info: hello.texi
    makeinfo hello.texi
hello.epub: hello.texi
    makeinfo --epub hello.texi
hello_html/index.html: hello.texi
    makeinfo --html hello.texi
```

Run make:

```
make
```

Read the docs with `calibre` or the `browser` or plain `info`:

```
calibre hello.epub & \
firefox hello_html/index.html & \
info -f ./hello.info
```

The HTML output is plain. You can adapt it with CSS by adding `--css-include=FILENAME` or `--css-ref=URL`.

Alternately you can write an Org Mode document and evaluate `(require 'ox-texinfo)` to activate exporting to texinfo.

43 Track changes with a version tracking system like `Mercurial` or `Git`

For convenience, first initialize a version tracking repository, for example Mercurial or Git.

```
# either Mercurial
hg init hello
# or Git
git init hello
# enter the repository folder
cd hello-scm/
```

Now you can add new files with

```
# in Mercurial
hg add FILE
# in Git
git add FILE
```

And take a snapshot of changes with

```
# in Mercurial
hg commit -m "a change description"
# in Git
git commit -a -m "a change description"
```

It is good practice to always use a version tracking system.

For additional information and how to publish your code if you want to, see the Mercurial Guide or the Git Tutorial.

44 Package with `autoconf` and `automake`

Create a `configure.ac` file with name, contact info and version.

```
dnl configure.ac
dnl Name, Version, and contact information.
AC_INIT([hello], [0.0.1], [myName@example.com])
# Find a supported Guile version and set it as @GUILE@
GUILE_PKG([3.0])
GUILE_PROGS
GUILE_SITE_DIR
AC_PREFIX_PROGRAM([guile])
AM_INIT_AUTOMAKE([gnu])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Add a `Makefile.am` with build rules. Only the start needs to be edited:

```
bin_SCRIPTS = hello # program name
SUFFIXES = .w .scm .sh
SCHEME = hello.scm # source files
hello: $(SCHEME)
    echo "#!/usr/bin/env bash" > "$@" && \
    echo 'exec -a "$$0" guile' \
      '-L "$$(dirname "$$(realpath "$$0")")"' \
      '-L "$$(dirname "$$(realpath
      ↪  "$$0")")/../share/guile/site/3.0/"' \
      '-s "$$0" "$$@"' \
      >> "$@" && echo ";; exec done: !#" >> "$@" && \
    cat "$<" >> "$@" && chmod +x "$@"
TEXINFO_TEX = doc/hello.texi # else it needs texinfo.texi
info_TEXINFOS = doc/hello.texi
# add library files, prefix nobase_ preserves directories
nobase_site_DATA =
```

The rest of the Makefile.am can be copied verbatim:

```
## Makefile.am technical details

# where to install guile modules to import. See
# https://www.gnu.org/software/automake/manual/html_node/Alte
↪  rnative.html
sitedir = $(datarootdir)/guile/site/$(GUILE_EFFECTIVE_VERSION)

GOBJECTS = $(nobase_site_DATA:%.w=%.go)
nobase_go_DATA = $(GOBJECTS)
godir=$(libdir)/guile/$(GUILE_EFFECTIVE_VERSION)/site-ccachep

# Make sure that the mtime of installed compiled files
# is greater than that of installed source files.  See:
# http://lists.gnu.org/archive/html/guile-devel/2010-07/msg00
↪  125.html
# The missing underscore before DATA is intentional.
guile_install_go_files = install-nobase_goDATA
$(guile_install_go_files): install-nobase_siteDATA

EXTRA_DIST = $(SCHEME) $(info_TEXINFOS) $(nobase_site_DATA)
CLEANFILES = $(GOBJECTS) $(wildcard *~)
DISTCLEANFILES = $(bin_SCRIPTS) $(nobase_site_DATA)

# precompile all source files
.scm.go:
    $(GUILE_TOOLS) compile $(GUILE_WARNINGS) \
       -o "$@" "$<"
```

```
## Makefile.am help

.PHONY: help
help: ## Show this help message.
	@echo 'Usage:'
	@echo ':make [target] ...' \
	    | sed "s/\(target\)/\\x1b[36m\1\\x1b[m/" \
	    | column -c2 -t -s :
	@echo
	@echo 'Custom targets:'
	@echo -e "$$(grep -hE '^\S+:.*##' $(MAKEFILE_LIST) \
	    | sed -e \
	      's/:.*##\s*/:/' -e \
	      's/^\(.\+\):\(.*\)/:\\x1b[36m\1\\x1b[m:\2/' \
	    | column -c2 -t -s :)"
	@echo
	@echo '(see ./configure --help for setup options)'
```

This assumes that the folder `hello` *uses a Version tracking system.*

```
## Makefile.am basic additional files
.SECONDARY: ChangeLog AUTHORS
ChangeLog: ## create the ChangeLog from the history
    echo "For user-visible changes, see the NEWS file" > "$@"
    echo >> "$@"
    if test -d ".git"; \
        then cd "$(dirname "$(realpath .git)")"  \
        && git log --date-order --date=short \
        | sed -e '/^commit.*$/d' \
        | awk '/^Author/ {sub(/\\$/,""); getline t; print $0
        ↪  t; next}; 1' \
        | sed -e 's/^Author: //g' \
        | sed -e \
          's/\(.*\)>Date:   \([0-9]*-[0-9]*-[0-9]*\)/\2
          ↪  \1>/g' \
        | sed -e 's/^\(.*\) \(\)\t\(.*\)/\3    \1    \2/g' \
          >> "$@"; cd -; fi
    if test -d ".hg"; \
        then hg -R "$(dirname "$(realpath .hg)")" \
           log --style changelog \
           >> "$@"; fi
AUTHORS: ## create the AUTHORS file from the history
    if test -d ".git"; \
        then cd "$(dirname "$(realpath .git)")"  \
          && git log --format='%aN' \
          | sort -u >> "$@"; cd -; fi
    if test -d ".hg"; \
        then hg -R "$(dirname "$(realpath .hg)")" \
            --config extensions.churn= \
            churn -t "{author}" >> "$@"; fi
```

Now create a `README` and a `NEWS` file:

```
#+title: Hello

A simple example project.

* Requirements

- Guile version 3.0.10 or later.

* Build the project

#+begin_src bash
,# Build the project
autoreconf -i && ./configure && make
,# Create a distribution tarball
autoreconf -i && ./configure && make dist
#+end_src

* License

GPLv3 or later.
```

```
hello 0.0.1

- initialized the project
```

And for the sake of this example, a simple `hello.scm` file:

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

44.1 Init a project with `hall`

To simplify the setup, start it by getting the tool guile-hall (named Hall) as described in the manual under Distributing Guile Code. Then create a new project:

```
hall init hello -a "My Name" \
                -e "myName@example.com" \
                --execute
cd hello && hall build --execute
```

Add –license to change the license; GPLv3 or later is the default.

Hall creates a `configure.ac` file with name, contact information and version, and a `Makefile.am` with build rules. It also automaticallyp adds `TEXINFO`-rules for the folder `doc/`.

Open `Makefile.am` and add `.w` to `SUFFIXES`.

Then add the rule

```
.scm.go:
    $(AM_V_GEN)$(top_builddir)/pre-inst-env \
      $(GUILE_TOOLS) compile \
      $(GUILE_TARGET) $(GUILE_WARNINGS) \
      -o "$@" "$<"
```

45 Deploy a project to users

Enable people to access your project as a webserver behind nginx, as clientside browser-app, or as Linux package (Guix tarball).

Browser: as webserver. *On the web no one knows you're a Scheme.*

Guile provides a webserver module. A minimal webserver:

```
(import (web server)
        (web request)
        (web response)
        (web uri))
(define (handler request body)
  (define path (uri-path (request-uri request)))
  (values (build-response
           #:headers `((content-type . (text/plain)))
           #:code 404)
          (string-append "404 not found: " path)))
(define v4 #t)
;; choose either IPv4 or IPv6; to suport both, run twice.
;; (run-server handler 'http
;;   (if v4 '(#:port 8081)
;;          '(#:family AF_INET6 #:port 8081)))
```

An nginx SSL Terminator (`/etc/nginx/sites-enabled/default`):

```
server {
  server_name domain.example.com;
  location / {
    proxy_pass http://localhost:8081;
  }
}
```

Set up SSL support with certbot (this edits the config file).

Browser again: clientside wasm. To run clientside, you can package your project with Hoot: build an interface and compile to wasm:

```
(use-modules (hoot ffi)) ;; guile-specific import
(define-foreign document-body "document" "body"
  -> (ref null extern))
(define-foreign make-text-node "document" "createTextNode"
  (ref string) -> (ref null extern))
(define-foreign append-child! "element" "appendChild"
  (ref null extern) (ref null extern)
  -> (ref null extern))

(append-child! (document-body) ;; Your logic
  (make-text-node "Hello, world!"))
```

Transpile with `guild compile-wasm`. If you run Guix:

```
guix shell guile-hoot guile-next -- \
  guild compile-wasm -o hoot.wasm hoot.scm
```

Get reflection tools from Guile Hoot (licensed Apache 2.0) with Guix:

```
guix shell guile-hoot guile-next --  bash -c \
 'cp $GUIX_ENVIRONMENT/share/guile-hoot/*/reflect*/{*.js,
 ↪  *.wasm}
 ↪  ./'
```

```
window.addEventListener("load", () =>
  Scheme.load_main("./hoot.wasm", {
    user_imports: { document: {
      body() { return document.body; },
      createTextNode: Document.prototype
        .createTextNode.bind(document)
    }, element: {
        appendChild(parent, child) {
        return parent.appendChild(child);}}}}));
```

Include `reflect.js` and `hoot.js` from a HTML page:

```
<html><head><title>Hello Hoot</title>
<script type="text/javascript" src="reflect.js"></script>
<script type="text/javascript" src="hoot.js"></script>
</head><body><h1>Hoot Test</h1></body></html>
```

For local testing, hoot provides a minimal webserver:

```
guix shell guile-hoot guile-next -- \
  guile -c '((@ (hoot web-server) serve))'
```

Linux: Guix tarball. *The package is the tarball.* — *Ludovic*

Guix can assemble a tarball of all dependencies. If you already have an autoconf project, this just requires a `guix.scm` file:

```
(import (gnu packages web)
        (gnu packages bash)
        (gnu packages guile)
        (gnu packages guile-xyz)
        (gnu packages pkg-config)
        (guix packages)
        (guix gexp)
        (guix build-system gnu)
        (prefix (guix licenses) license:))

(define-public guile-doctests
  (package
   (name "guile-doctests")
   (version "0.0.1")
   (source (local-file "." "" #:recursive? #t))
   (build-system gnu-build-system)
   (propagated-inputs `(("guile" ,guile-3.0)
                        ("pkg-config" ,pkg-config)
                        ("bash" ,bash)))
   (home-page "https://hg.sr.ht/~arnebab/guile-doctests")
   (synopsis "Tests in procedure definitions")
   (description "Guile module to keep tests directly in
   ↪  your procedure definition.")
   (license license:lgpl3+)))

guile-doctests
```

First test building `guix build -f guix.scm`, then test running with `guix shell -f guix.scm` and once both work, create your package with:

```
guix pack -e '(load "guix.scm")' \
     -RR -S /bin=bin -S /share=share
```

Copy the generated tarball. In can be executed with:

```
mkdir hello && cd hello && tar xf TARBALL_FILE && \
  bin/doctest
```

Since this tarball generation is a bit finicky, there is a guile-doctests package with a working example setup.

Once you have `guix pack` working, you can also create `dockerfiles` and other packages to deploy into different publishing infrastructure.

To be continued: Scheme is in constant development and deploying Guile programs is getting easier. Lilypond solved Windows.

Also see the Map of R^{7}RS and the Scheme primer to keep learning.

You are ready.

Go and build a project you care about.

List of Links

Get the gist of Lisp in practical steps.

This book guides you into **the heart of programming** with Scheme, to give you a smooth start into one of the oldest standardized and thriving languages.

We are the namegivers,
the dreamers who build tools of sand and logic
to **surpass the limits of our minds**.

Choose your path
through **a map of building blocks**
to take on challenges by code.