Small snippets of Guile Scheme

There's a lot of implicit knowledge among Guile developers. Here I gather some useful snippets I found along the way.

More useful stuff to get things done in Guile is available in guile-basics and py2guile.

Contents

1 log-expression: print variable name and value

During debugging I often want to display variables by name and value. I used to print name and value by hand, but this quickly becomes tedious:

```
(define foo 'bar)
(format #t "foo: ~a\n" foo)
;; or
(display 'foo)(display foo)(newline)
```

Therefore I build me something simpler:

```
(define-syntax-rule (log-exprs exp ...)
  (begin (format #t "~a: ~S\n" (quote exp) exp) ...))
```

Now I can simply log variables like this:

```
(define foo 'bar)
(log-exprs foo)
;; => foo: bar
(define bar 'baz)
(log-exprs foo bar (list "hello"))
;; foo: bar
;; bar: baz
;; (list hello): ("hello")
```

2 Use Guile-yaml to search for the first match in a yaml file

This uses guile-libyaml to parse the yaml file and `(ice-9 match)` to recursively search within the file.

The example file is demo1.yml from the guile-yaml repo:

```
---
 doe: "a deer, a female deer"
 ray: "a drop of golden sun"
 pi: 3.14159
 xmas: true
 french-hens: 3
 calling-birds:
   - huey
```

```
   - dewey
   - louie
   - fred
 xmas-fifth-day:
   calling-birds: four
   french-hens: 3
   golden-rings: 5
   partridges:
     count: 1
     location: "a pear tree"
   turtle-doves: two
```

I'm searching for the first match to "partridges":

```
(import (yaml) (ice-9 match))
(define demo (read-yaml-file "demo1.yml"))
(let match-demo ((demo demo))
  (match demo
    ((("partridges" . b) c ...) b)
    (else (if (pair? demo)
              (or (match-demo (cdr demo)) (match-demo (car demo)))
              #f))))
;; => (("count" . "1") ("location" . "a pear tree"))
```

To use this snippet, start guile as `guile -L .` in the guile-libyaml repo.

3 Count occurences of each key in an alist

This was asked by *fnstudio* in the #guile channel on Freenode.

Given a list of pairs (like xy-coordinates), count how often the first element appears.

```
(define xydata '((1 . 1)(1 . 2)(1 . 3)(2 . 1)(2 . 2)))

(define (assoc-increment! key alist)
  "Increment the value of the key in the alist, set it to 1 if it does
  ↪  not exist."
  (define res (assoc key alist))
  (assoc-set! alist key (or (and=> (and=> res cdr) 1+) 1)))

(fold assoc-increment! '() (map car xydata))
;; => ((2 . 2) (1 . 3))
```

4 Common Substrings and Somewhat Cheap String Similarity

Getting the longest common substring is a traditional task, a simplification of actual edit distance like the Levenshtein distance (which actually has fast estimators).

But maybe you want all common substrings without duplicates.

The following code is not optimized, but it works.

4.1 Get all common substrings

4.1.1 Usage

Different from the requirements in in Rosetta Code, this returns not the longest common substring, but all non-consecutive substrings.

```
(longest-common-substrings "thisisatest" "testing123testing")
;; => ("test" "i")
(longest-common-substrings "thisisatestrun" "thisisatestunseen")
;; => ("thisisatest" "un")
```

4.1.2 Implementation

Warning: This is NOT fast. It takes a few seconds when run over two text documents with around 2000 characters each.

```
(import (srfi srfi-1))
(define (longest-common-substrings s1 s2)
  (define c1 (string->list s1))
  (define c2 (string->list s2))
  (define (common-prefix a b)
    (let loop ((prefix '()) (a a) (b b))
      (cond ((or (null? a) (null? b)) (reverse! prefix))
            ((not (equal? (car a) (car b))) (reverse! prefix))
            (else (loop (cons (car a) prefix) (cdr a) (cdr b))))))
  (define (longer a b)
    (if (> (length a) (length b))
        a b))
  (define (common-substrings a b)
    (define substrings '())
    (let loop ((a2 a) (b b) (longest '()))
      (let ((prefix (common-prefix a2 b)))
        (let ((anew (drop a2 (length prefix))))
```

```
          (when (not (null? prefix))
            (let ((str (apply string prefix)))
              (when (not (member str substrings))
                (set! substrings (cons str substrings)))))
          (cond ((null? b) #t) ;; done
                ((null? anew)
                 (loop a (cdr b) '()))
                ((null? prefix)
                 (loop (cdr anew) b longest))
                (else
                 (loop (cdr anew) b (longer prefix longest)))))))
    substrings)
  (let ((substrings (common-substrings c1 c2)))
    (define (contained-in-any-other s)
      (any identity
           (map (λ (s2) (and (not (equal? s s2)) (string-contains s2
           ↪  s)))
                substrings)))
    (reverse! (remove contained-in-any-other substrings))))
```

4.2 Somewhat Cheap String Similarity

4.2.1 Usage

```
(cheap-similarity "thisisatest" "thisisatest")
;; => 3.045454545454454
(cheap-similarity "thisisatest" "thisisatestrun")
;; => 2.68
(cheap-similarity "thisisatest" "testing123testing")
;; => 1.2142857142857142
(cheap-similarity "thisisatest" "criecriecriecrie")
;; => 0.444444444444444
```

4.2.2 Implementation

Warning: This is NOT well defined and is mostly **untested**, so it might have ugly unforseed edge-cases and might give bad results for large classes of problems. It seems to do what I want (get the similarity of filenames for ordering streams by similarity for the guile media site), and it was interesting no write, but that't about it. **Use with caution!**

```
(import (srfi srfi-1))
```

```
(define (all-common-substrings s1 s2)
  (define c1 (string->list s1))
  (define c2 (string->list s2))
  (define (common-prefix a b)
    (let loop ((prefix '()) (a a) (b b))
      (cond ((or (null? a) (null? b)) (reverse! prefix))
            ((not (equal? (car a) (car b))) (reverse! prefix))
            (else (loop (cons (car a) prefix) (cdr a) (cdr b))))))
  (define (longer a b)
    (if (> (length a) (length b))
        a b))
  (define (common-substrings a b)
    (define substrings '())
    (let loop ((a2 a) (b b) (longest '()))
      (let ((prefix (common-prefix a2 b)))
        (let ((anew (drop a2 (length prefix))))
          (when (not (null? prefix))
            (let ((str (apply string prefix)))
              (set! substrings (cons str substrings))))
          (cond ((null? b) #t) ;; done
                ((null? anew)
                 (loop a (cdr b) '()))
                ((null? prefix)
                 (loop (cdr anew) b longest))
                (else
                 (loop (cdr anew) b (longer prefix longest)))))))
    substrings)
  (let ((substrings (common-substrings c1 c2)))
    (define (contained-in-any-other s)
      (any identity
           (map (λ (s2) (and (not (equal? s s2)) (string-contains s2
           ↪  s)))
                substrings)))
    substrings))

(define (cheap-similarity s1 s2)
  (define common-length
    (apply + (map string-length (all-common-substrings s1 s2))))
  (define total-length (+ (string-length s1) (string-length s2)))
  (/ common-length total-length 1.))
```

5 pivot a table

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

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

Explanation: Apply moves `map list` into the list, so this changes to

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

Map takes the elements from each of the arguments to call the function, so the result is

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

Or in shorthand notation:

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

[2023-07-05 Mi]

6 Script with minimal startup time

To create a script that starts fast, you'll want to avoid the parsing time.

If you have a script in a file named `hello-world.scm`, use the following:

```
#!/usr/bin/env bash
exec -a "$0" guile -L $(dirname $(realpath "$0")) \
                   -C $(dirname $(realpath "$0")) \
                   -e '(hello-world)' -c '' "${@}"
# terminate the inline comment "started" with the #! line: !#
(define-module (hello-world)
  #:export (main))

(define (main args)
  (format #t "Hello ~a\n" (cdr args)))
```

Then make it executable and run it as script:

```
chmod +x hello-world.scm
./hello-world.scm World
# => Hello (World)
```

As a benchmark, run it 100 times:

```
# exclude single-time auto-compilation
./hello-world.scm World >/dev/null
{ time for i in {1..100}; do
./hello-world.scm World > /dev/null
done } 2>&1
```

I get:

```
real 0m1,601s
user 0m0,944s
sys 0m0,721s
```

So on my machine, startup time is 16ms.

[2024-01-11 Do]

7 Shell scripts in Guile

Similar to the script above, but with a bit more convenience.

You can write shell scripts in Guile, but you'll need to make sure you to do it in a way which has low startup times.

The fastest I found which stays fast with larger scripts is shell-indirection calling your scripts as modules:

```
====== scriptname.scm ======
```

```
#!/usr/bin/env bash
# -*- scheme -*-
# set Guile if unset
if [ -z ${GUILE+x} ]; then
    GUILE=guile
fi

exec -a "$0" "${GUILE}" -L $(dirname $(realpath "$0")) -e
↪  '(scriptname)' -c '' "$@"
; !#

(define-module (scriptname) ;; same as filename!
   #:export (main))

(define (main args)
   (display args)
```

```
  (newline))
```

Copy this into `~/.local/bin/`, make it executable with `chmod +x ~/.local/bin/scriptname.scm`, and you have powerful scripting with low startup timme.

```
time for i in {1..100}; do scriptname.scm >/dev/null ; done
real        0m3,202s
user        0m2,683s
sys        0m1,110s
```

⇒ `32ms` runtime. Not as fast as perl or bash (calling bash from bash just takes 3ms), but pretty good.

[2024-04-18 Do]

8 define-typed

If you want to add typechecks, you can follow the format by sph-sc, a Scheme to C compiler. It declares types after the function definition like this:

```
(define (hello typed-world) (string? string?)
  typed-world)
```

That's simple enough that a plain, hygienic `syntax-rule` can support it.

8.1 Implementation

For performance reasons, the following defines `define-typed` and `define-typed*`, where `define-typed*` supports `#:keyword` arguments:

```
(define-module (define-typed) #:export (define-typed* define-typed))

(import (srfi :11 let-values))

;; common procdures
(define-inlinable (call-and-check-return-type proc ret?)
  (if ret? ;; #f means: do not check
      ;; get the result
      (let-values ((res (proc)))
        ;; typecheck the result
        (unless (apply ret? res)
          (error "type error: return value ~a does not match ~a"
```

```
                  res ret?))
        ;; return the result
        (apply values res))
      (proc)))

(define-inlinable (check-argument-and-type-count args types)
  (unless (equal? (length args) (length types))
    (error "argument error: number of arguments ~a and types ~a differs"
      args types)))

(define (add-properties! proc name from-proc ret? types)
  ;; add procedure properties via an inner procedure
  (set-procedure-properties! proc (procedure-properties from-proc))
  ;; record the types
  (set-procedure-property! proc 'return-type ret?)
  (set-procedure-property! proc 'argument-types types)
  ;; preserve the name
  (set-procedure-property! proc 'name name))


;; specific to define-typed
(define-syntax check-types
  (syntax-rules ()
    ((_ (type? types? ...) (argument arguments ...))
     (begin
       (unless (type? argument)
         (error "type error ~a ~a" type? argument))
       (check-types (types? ...) (arguments ...))))
    ((_ () ()) #f)))

(define-syntax-rule
  (define-typed (procname args ...) (ret? types ...) body ...)
  "Define a procedure with typechecks."
  (begin
    (define properties-helper (lambda (args ...) body ...))
    (define (procname args ...)
      ;; create a sub-procedure to run after typecheck
      (define (inner)
        body ...)
      ;; typecheck the arguments
      (check-types (types ...) (args ...))
      ;; get and check the result
      (call-and-check-return-type inner ret?))
    (check-argument-and-type-count
```

```
      (quote (args ...)) (quote (types ...)))
    ;; add properties and return the inner procedure
    (add-properties! procname 'procname properties-helper
                     ret? (list types ...))))


;; specific to define-typed*
(define-syntax check-types*
  (syntax-rules ()
    ((_ (type? types? ...) (argument arguments ...))
     (begin
       (if (and (keyword? type?)
                (keyword? argument))
           (unless (equal? type? argument)
             (error "Keywords in arguments and types differ ~a ~a"
                    type? argument))
           (unless (type? argument)
             (error "type error ~a ~a" type? argument)))
       (check-types* (types? ...) (arguments ...))))
    ((_ () ()) #f)))

(define-syntax-rule
  (define-typed* (procname args ...) (ret? types ...) body ...)
  "Define a procedure with typecheck, taking keywords into account."
  (begin
    (define properties-helper (lambda* (args ...) body ...))
    (define* (procname args ...)
      ;; create a sub-procedure to run after typecheck
      (define (inner)
        body ...)
      ;; typecheck the arguments
      (check-types* (types ...) (args ...))
      ;; get and check the result
      (call-and-check-return-type inner ret?))
    (check-argument-and-type-count
      (quote (args ...)) (quote (types ...)))
    ;; add properties and return the inner procedure
    (add-properties! procname 'procname properties-helper
                     ret? (list types ...))))
```

This supports most features of regular define like docstrings, procedure properties, multiple values (thanks to Vivien!), and so forth.

`define-typed*` also supports keyword-arguments (thanks to Zelphir Kaltstahl's contracts), but is slower.

8.2 Usage

```
(define-typed (hello typed-world) (string? string?)
  typed-world)
(hello "typed")
;; => "typed"
(hello 1337)
;; => type error ~a ~a #<procedure string? (_)> 1337
(define-typed (hello typed-world) (string? string?)
  "typed" ;; docstring
  #((props)) ;; more properties
  1337) ;; wrong return type
(procedure-documentation hello)
;; => "typed"
(procedure-properties hello)
;; =>((argument-types #<procedure string? (_)>)
;;    (return-type . #<procedure string? (_)>)
;;    (name . hello) (documentation . "typed") (props))
(hello "typed")
;; type error: return value ~a does not match ~a (1337) #<procedure
↪  string? (_)>
```

Multiple Values and optional and required keyword arguments:

```
(define-typed (multiple-values num) ((λ(a b) (> a b)) number?)
  (values (* 2 (abs num)) num))
(multiple-values -3)
;; => 6
;; => -3
(define-typed* (hello #:key typed-world) (string? #:key string?)
  "typed" #((props)) typed-world)
(hello #:typed-world "foo")
;; => "foo"
;; unused keyword arguments are always boolean #f as input
(hello)
;; => type error ~a ~a #<procedure string? (_)> #f
;; typing optional keyword arguments
(define (optional-string? x) (or (not x) (string? x)))
(define-typed* (hello #:key typed-world) (string? #:key
↪  optional-string?)
  (or typed-world "world"))
(hello)
;; => "world"
(hello #:typed-world "typed")
```

```
;; => "typed"
(hello #:typed-world #t)
;; => type error ~a ~a #<procedure optional-string? (x)> #t
;; optional arguments
(define-typed* (hello #:optional typed-world) (string? #:optional
↪  optional-string?)
  (or typed-world "world"))
(hello)
;; => "world"
(hello "typed")
;; => "typed"
(hello #t)
;; => type error ~a ~a #<procedure optional-string? (x)> #t
```

8.3 Benchmark

`define-typed` automates some of the guards of Optimizing Guile Scheme, so the compiler can optimize more (i.e. if you check for `real?`) but keep in mind that these checks are not free: use typechecks outside tight loops, except where they provably provide an improvement.

```
#!/usr/bin/env bash
exec guile -L . "$0"
; !#
(import (define-typed) (statprof))

;; The primitive procedure
(define (magnitude x y) (sqrt (+ (* x x) (* y y))))
;; The hand-optimized procedure from Optimizing Guile Scheme
(define (magnitude-handtyped x y)
  (unless (and (real? x) (inexact? x) (real? y) (inexact? y))
    (error "expected floats" x y))
  (sqrt (+ (* x x) (* y y))))

;; Typing helper (inlinable to prevent indirection costs)
(define-inlinable (float? x) (and (real? x) (inexact? x)))

;; Fast define-typed
(define-typed (magnitude-typed x y) (#f float? float?)
  (sqrt (+ (* x x) (* y y))))

;; slower define-typed* with keyword support
```

```
(define-typed* (magnitude-typed* x y #:key foo) (#f float? float? #:key
↪  not)
  (sqrt (+ (* x x) (* y y))))

(statprof
 (λ _
   (let lp ((i 0))
     (when (< i 20000000)
       (magnitude 3.0 4.0)
       (lp (+ i 1))))))

(statprof
 (λ _
   (let lp ((i 0))
     (when (< i 20000000)
       (magnitude-handtyped 3.0 4.0)
       (lp (+ i 1))))))

(statprof
 (λ _
   (let lp ((i 0))
     (when (< i 20000000)
       (magnitude-typed 3.0 4.0)
       (lp (+ i 1))))))


(statprof
 (λ _
   (let lp ((i 0))
     (when (< i 20000000)
       (magnitude-typed* 3.0 4.0)
       (lp (+ i 1))))))
```

Results:

```
%     cumulative   self
time   seconds     seconds  procedure
 94.12      2.12      1.99  benchmark.scm:6:0:magnitude
  5.88      0.12      0.12  %after-gc-thunk
  0.00      2.12      0.00  benchmark.scm:14:1
  0.00      0.12      0.00  anon #x36ac5070
---
Sample count: 51
Total time: 2.119640515 seconds (1.915531414 seconds in GC)
```

```
%     cumulative   self
time   seconds     seconds  procedure
 94.12      0.56      0.56  benchmark.scm:7:0:magnitude-handtyped
  5.88      0.59      0.03  benchmark.scm:21:1
---
Sample count: 17
Total time: 0.592612292 seconds (0.440981526 seconds in GC)
%     cumulative   self
time   seconds     seconds  procedure
100.00      0.59      0.59  benchmark.scm:10:0:magnitude-typed
  0.00      0.59      0.00  benchmark.scm:28:1
---
Sample count: 13
Total time: 0.589839272 seconds (0.390181781 seconds in GC)
%     cumulative   self
time   seconds     seconds  procedure
 94.83      2.24      2.16  benchmark.scm:11:0:magnitude-typed*
  3.45      0.08      0.08  %after-gc-thunk
  1.72      2.28      0.04  benchmark.scm:36:1
  0.00      0.08      0.00  anon #x36ac5070
---
Sample count: 58
Total time: 2.280908715 seconds (1.992148184 seconds in GC)
```

`define-typed` reaches the performance of the hand-optimized procedure magnitude-handtyped while `define-typed*` is as fast as an untyped procedure in this case where constraining types provides a big benefit.

8.4 Summary

Typecheks from `define-typed` provide a type boundary instead of forcing explicit static typing.

Also you can do more advanced checks by providing your own test procedures and validating your API more elegantly, but these then may not help the compiler produce faster code.

But keep in mind that this does not actually provide static program analysis like while-you-write type checks. It's simply syntactic sugar for a boundary through which only allowed values can pass. Thanks to program flow analysis by the just-in-time compiler, it can make your code faster, but that's not guaranteed. It may be useful for your next API definition.

define-typed: a static type syntax-rules macro for Guile to create API contracts and help the JIT compiler create more optimized code.

[2024-05-09 Do]

9 Guile imports in Makefiles

Use modules in the guile-session of Makefiles:

```
output: .guile-session
    touch $(guile (list-ec (: i 2) "$@"))
# => touch output output

.PHONY: .guile-session
.guile-session:
    $(guile (import (srfi :42 eager-comprehensions)))
```

The import runs only once, but is guaranteed to run at every execution (due to `.PHONY`).

Importing `srfi :42` provides Eager Comprehensions which are similar to Python's list-comprehensions.

[2024-05-28 Di]

10 minimal syntax rule that uses homoiconicity

```
(define-syntax-rule (writer (code ...))
   (apply write (cdr (list code ...))))
(writer (display "Hello World"))
;; => "Hello World"
;; while
(display "Hello World")
;; => Hello World
;;     ^ no quotes
```

[2025-01-17 Fr]

11 Generator functions with state

A generator is no magic: It just uses `let` to create a scope with a variable and then defines a procedure that can access the scope.

```
(define (counter-init)
  (let ((i 0))
    (define (counter)
      (set! i (1+ i))
      i)
    counter))
(define a-counter (counter-init))
(list
  (a-counter)
  (a-counter)
  (a-counter))
```

1 2 3

[2025-05-07 Mi]