# Small snippets of Guile Scheme

Dr. Arne Babenhauserheide

*<2020-08-25 Di>*

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

## 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")
```

## 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.

## 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))
```

## 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.

- Get all common substrings
  - 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")
    ```

  - 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))
          ;; (format #t "a2 ~a\nanew ~a\nb ~a\n\n" (apply string a2) (appl
          (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 s
  (reverse! (remove contained-in-any-other substrings))))
```

- Somewhat Cheap String Similarity
  - Usage

    ```
    (cheap-similarity "thisisatest" "thisisatest")
    ;; => 3.0454545454545454
    (cheap-similarity "thisisatest" "thisisatestrun")
    ;; => 2.68
    (cheap-similarity "thisisatest" "testing123testing")
    ;; => 1.2142857142857142
    (cheap-similarity "thisisatest" "criecriecriecrie")
    ;; => 0.4444444444444444
    ```

  - 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 s
    substrings))

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

# Benchmarking Guile Versions with the R7RS Benchmarks

To test changes in speed between different guile Versions, I use the R7RS benchmarks by ecraven as well as my own evaluation to get the geometric mean of the slowdown compared to the fastest. How to reproduce:

```
hg clone https://hg.sr.ht/~arnebab/wisp # needs https://mercurial-scm.org
git clone https://git.savannah.gnu.org/git/guile.git # needs https://git-scm.com
git clone https://github.com/ecraven/r7rs-benchmarks
cd guile && git checkout main # replace main with the revision to test
guix environment guile # opens a shell, needs to run on https://guix.gnu.org
guix shell gperf # needed to build guile from shell
./autogen.sh && ./configure --prefix=$HOME/.local && make
cd ../r7rs-benchmarks
rm results.Guile
VERSION=3
guix shell guile -- bash -c \
  'GUILD=../guile/meta/guild GUILE=../guile/meta/guile ./bench guile all'
sed -i s/guile-/guile-${VERSION}-/g results.Guile
sed -i s/guile,/guile-${VERSION},/g results.Guile
mv results.Guile results.Guile${VERSION}
# repeat for other versions of Guile (git checkout + VERSION=...)
grep CSVLINE results.Guile* | sed 's,+!CSVLINE!+,,' >  /tmp/all.csv
cd ../wisp/examples
for i in 3; do ./evaluate-r7rs-benchmark.w /tmp/all.csv guile-$i; done
```

An example run:

```
for i in 2 3 jit nojit; do
  ./evaluate-r7rs-benchmark.w /tmp/all.csv guile-$i 2>/dev/null
done | grep -A2 "Geometric Mean" 2>/dev/null
```

Results of just one run (these are no representative statistics!):

```
=== Guile-2 Geometric Mean slowdown (successful tests / total tests) ===

2.8389855923409266 (56 / 57)

=== Guile-3 Geometric Mean slowdown (successful tests / total tests) ===

1.395836097066007 (56 / 57)

=== Guile-Jit Geometric Mean slowdown (successful tests / total tests) ===

1.0074935538381193 (56 / 57)
```

```
=== Guile-Nojit Geometric Mean slowdown (successful tests / total tests) ===

3.0961877404441847 (56 / 57)
```

Guile Jit and Guile Nojit are runs with just-in-time compilation forced (Jit: ) or disabled (Nojit). See GUILE_JIT_THRESHOLD in the handbook.

(This should not be interpreted as recommendation to always force the JIT. In normal operation letting Guile decide when to JIT-compile should provide a better tradeoff than basing such decisions on synthetic benchmark results)

*[2022-11-19 Sa]*