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) e
```

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))
```