

金鱼 Scheme 文学编程

目录

vertical_space_before	9
1 概述	9
vertical_space_before	11
2 boot.scm	11
vertical_space_before	15
3 (liii base)	15
3.1 许可证	15
3.2 接口	16
3.3 测试	16
3.4 表达式	18
3.4.1 原语表达式	18
3.4.2 派生表达式	18
3.5 程序结构	21
3.6 相等性判断	22
3.7 数	22
3.7.1 数的类型	22
3.7.2 数的准确性	22
3.7.3 数的运算	22
3.7.4 数的转换	27
3.8 布尔值	27
3.9 序对和列表	28
3.10 字符	28
3.11 字符串	28
3.12 向量	29
3.13 字节向量	29
3.14 控制流	29
3.15 异常处理	29
3.16 输入和输出	31
3.16.1 端口	31
3.16.2 输入	32
3.16.3 输出	32
3.17 系统接口	32
3.18 三鲤扩展库	32
3.19 结尾	36
vertical_space_before	37
4 (liii error)	37
4.1 许可证	37
4.2 接口	38
4.3 测试	38
4.4 实现	38
4.5 结尾	40
vertical_space_before	41
5 (liii check)	41
5.1 协议	41
5.2 接口	42
5.3 实现	42
5.4 结尾	48
vertical_space_before	51
6 (liii case)	51

6.1 测试	51
6.2 结尾	52
<div>vertical_space_before</div> 7 (liii string)	<div>no_line_break</div> 53
7.1 许可证	<div>no_line_break</div> 53
7.2 接口	<div>no_line_break</div> 54
7.3 测试	<div>no_line_break</div> 55
7.4 内部公共子函数	<div>no_line_break</div> 56
7.5 谓词	<div>no_line_break</div> 56
7.6 列表-字符串转换	<div>no_line_break</div> 58
7.7 谓词	<div>no_line_break</div> 62
7.8 选择器	<div>no_line_break</div> 63
7.9 前缀和后缀	<div>no_line_break</div> 64
7.10 搜索	<div>no_line_break</div> 74
7.11 翻转和追加	<div>no_line_break</div> 76
7.12 Fold, unfold & map	<div>no_line_break</div> 77
7.13 插入和解析	<div>no_line_break</div> 78
7.14 结尾	<div>no_line_break</div> 79
<div>vertical_space_before</div> 8 (liii list)	<div>no_line_break</div> 81
8.1 许可证	<div>no_line_break</div> 81
8.2 接口	<div>no_line_break</div> 82
8.3 测试	<div>no_line_break</div> 83
8.4 SRFI-1	<div>no_line_break</div> 83
8.4.1 构造器	<div>no_line_break</div> 83
8.4.2 谓词	<div>no_line_break</div> 84
8.4.3 选择器	<div>no_line_break</div> 86
8.4.4 常用函数	<div>no_line_break</div> 92
8.4.5 折叠和映射	<div>no_line_break</div> 93
8.4.6 过滤和分组	<div>no_line_break</div> 94
8.4.7 搜索	<div>no_line_break</div> 98
8.4.8 删除	<div>no_line_break</div> 101
8.5 三鲤扩展库	<div>no_line_break</div> 103
8.6 结尾	<div>no_line_break</div> 107
<div>vertical_space_before</div> 9 (liii vector)	<div>no_line_break</div> 109
9.1 许可证	<div>no_line_break</div> 109
9.2 接口	<div>no_line_break</div> 110
9.3 测试	<div>no_line_break</div> 111
9.4 实现	<div>no_line_break</div> 111
9.4.1 构造器	<div>no_line_break</div> 111
9.4.2 谓词	<div>no_line_break</div> 113
9.4.3 选择器	<div>no_line_break</div> 114
9.4.4 迭代	<div>no_line_break</div> 114
9.4.5 搜索	<div>no_line_break</div> 115
9.4.6 修改器	<div>no_line_break</div> 117
9.4.7 转换	<div>no_line_break</div> 121
9.5 结尾	<div>no_line_break</div> 123
<div>vertical_space_before</div> 10 (liii stack)	<div>no_line_break</div> 125
10.1 许可证	<div>no_line_break</div> 125
10.2 接口	<div>no_line_break</div> 125
10.3 测试	<div>no_line_break</div> 126
10.4 实现	<div>no_line_break</div> 126
10.5 结尾	<div>no_line_break</div> 127

vertical_space_before		7
11 (liii queue)		129
11.1 许可证		129
11.2 接口		129
11.3 测试		129
11.4 实现		131
11.5 结尾		132
vertical_space_before		
12 (liii comparator)		133
12.1 许可证		133
12.2 接口		134
12.3 测试		136
12.4 内部函数		138
12.5 构造器		137
12.6 标准函数		141
12.7 比较谓词		144
12.8 结尾		144
vertical_space_before		
13 (liii hash-table)		145
13.1 许可证		145
13.2 测试		146
13.3 接口		149
13.3.1 访问哈希表中的元素		147
13.4 实现		147
13.4.1 子函数		147
13.4.2 构造器		147
13.4.3 谓词		149
13.4.4 选择器		149
13.4.5 修改器		150
13.4.6 哈希表整体		152
13.4.7 映射和折叠		154
13.4.8 复制和转换		155
13.5 结尾		155
vertical_space_before		
14 (liii set)		157
14.1 许可证		157
14.2 接口		158
14.3 结尾		158
vertical_space_before		
15 三鳉扩展库说明		159
15.1 结尾		159
vertical_space_before		
16 (scheme case-lambda)		161
16.1 协议		161
16.2 接口		161
16.3 实现		162
vertical_space_before		
17 (scheme char)		163
17.1 许可证		163
17.2 接口		163
17.3 实现		163
17.4 结尾		164
vertical_space_before		
18 (scheme file)		165
18.1 许可证		165

18.2 接口	165
18.3 实现	165
18.4 结尾	166
19 (srfi sicp)	167
19.1 许可证	167
19.2 接口	168
19.3 测试	168
19.4 实现	169
19.5 结尾	169
索引	171

no_line_break	165
no_line_break	165
no_line_break	165
no_line_break	166
no_line_break	167
no_line_break	167
no_line_break	167
no_line_break	168
no_line_break	168
no_line_break	168
no_line_break	169
no_line_break	169
no_line_break	169
no_line_break	171

第 1 章

概述

金鱼 Scheme，又称 Goldfish Scheme，是三鲤网络发起的 Scheme 解释器实现。目前专注于 Scheme 语言的标准库。本文档是金鱼 Scheme V17.10.6 的文学编程实现，目前金鱼 Scheme 中几乎所有的 Scheme 代码实现和测试都包含在本文档中。

本文档的用户是人类和大模型服务。基于本文档，用户可以获得金鱼 Scheme 的实现方法，也可以改进金鱼 Scheme 的既有实现。本文档的内容越翔实，用户使用金鱼 Scheme 的体验越好。

文档许可证暂时没有确定！

第 2 章

boot.scm

S7 Scheme默认并不遵循R7RS，在启动S7 Scheme之后，我们需要做的第一件事就是加载`boot.scm`，实现R7RS的`define-library`和`import`。

[R7rs](#) [索引](#)
file-exists?

goldfish/scheme/boot.scm

1 ▾

```
(define (file-exists? path)
  (if (string? path)
      (if (not (g_access path 0)) ; F_OK
          #f
          (if (g_access path 4) ; R_OK
              #t
              (error 'permission-error (string-append "No permission: " path))))
      (error 'type-error "(file-exists? path): path should be string"))
```

[R7rs](#) [索引](#)
delete-file

goldfish/scheme/boot.scm

△ 2 ▽

```
(define (delete-file path)
  (if (not (string? path))
      (error 'type-error "(delete-file path): path should be string")
      (if (not (file-exists? path))
          (error 'read-error (string-append path " does not exist"))
          (g_delete-file path))))
```

  **define-library**

goldfish/scheme/boot.scm

△ 3 ▽

```
; 0-clause BSD
; Adapted from S7 Scheme's r7rs.scm
(define-macro (define-library libname . body) ; |(lib name)| -> environment
  '(define ,(symbol (object->string libname))
    (with-let (sublet (unlet)
      (cons 'import import)
      (cons '*export* ())
      (cons 'export (define-macro (, (gensym) . names)
        '(set! *export* (append ',names *export*)))))
      ,@body
      (apply inlet
        (map (lambda (entry)
          (if (or (member (car entry) '(*export* export import))
              (and (pair? *export*)
                  (not (member (car entry) *export*))))
              (values)
              entry)))
          (curlet))))))

(unless (defined? 'r7rs-import-library-filename)
  (define (r7rs-import-library-filename libs)
    (when (pair? libs)
      (let ((lib-filename (let loop ((lib (if (memq (caar libs) '(only except prefix rename)
          (cadar libs)
          (car libs)))
        (name ""))
        (set! name (string-append name (symbol->string (car lib))))
        (if (null? (cdr lib))
            (string-append name ".scm")
            (begin
              (set! name (string-append name "/"))
              (loop (cdr lib) name))))))
        (unless (member lib-filename (*s7* 'file-names))
          (load lib-filename)))
        (r7rs-import-library-filename (cdr libs))))))
```

  **import**

goldfish/scheme/boot.scm

△ 4

```

(define-macro (import . libs)
  '(begin
    (r7rs-import-library-filename ',libs)
    (varlet (curlet)
      ,@(map (lambda (lib)
        (case (car lib)
          ((only)
            '((lambda (e names)
              (apply inlet
                (map (lambda (name)
                  (cons name (e name)))
                  names)))
              (symbol->value (symbol (object->string (cadr ',lib))))
              (caddr ',lib)))
          ((except)
            '((lambda (e names)
              (apply inlet
                (map (lambda (entry)
                  (if (member (car entry) names)
                      (values)
                      entry))
                  e)))
              (symbol->value (symbol (object->string (cadr ',lib))))
              (caddr ',lib)))
          ((prefix)
            '((lambda (e prefix)
              (apply inlet
                (map (lambda (entry)
                  (cons (string->symbol
                        (string-append (symbol->string prefix)
                                      (symbol->string (car entry))))
                        (cdr entry)))
                  e)))
              (symbol->value (symbol (object->string (cadr ',lib))))
              (caddr ',lib)))
          ((rename)
            '((lambda (e names)
              (apply inlet
                (map (lambda (entry)
                  (let ((info (assoc (car entry) names)))
                    (if info
                      (cons (cadr info) (cdr entry))
                      entry)))
                  e)))
              (symbol->value (symbol (object->string (cadr ',lib))))
              (caddr ',lib)))
          (else
            '((let ((sym (symbol (object->string ',lib))))
              (if (not (defined? sym))
                  (format () "~A not loaded~%" sym)
                  (symbol->value sym))))))
      libs))))

```

第 3 章

(liii base)

Goldfish Scheme解释器默认会加载(liii base)。如果不想默认加载(liii base)，请使用s7、r7rs或者sicp模式。

(liiii base)由以下函数库组成：

(scheme base). 由R7RS定义的Scheme基础函数库

(srfi srfi-2). 由SRFI-2定义的and-let*

(srfi srfi-8). 由SRFI-8定义的receive

(liiii base). 三鲤扩展函数和S7内置的非R7RS函数，例如display*

3.1 许可证

goldfish/scheme/base.scm

1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

goldfish/liiii/base.scm

1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

[tests/goldfish/liii/base-test.scm](#)

1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

3.2 接口

[goldfish/scheme/base.scm](#)

△ 2 ▾

```
(define-library (scheme base)
  (export
    let-values
    define-record-type
    square
    exact inexact
    floor s7-floor ceiling s7-ceiling truncate s7-truncate round s7-round
    floor-quotient
    gcd lcm s7-lcm
    boolean=?
    ; String
    string-copy
    ; Vector
    vector->string string->vector
    vector-copy vector-copy! vector-fill!
    ; Input and Output
    call-with-port port? binary-port? textual-port?
    input-port-open? output-port-open?
    open-binary-input-file open-binary-output-file
    close-port
    eof-object
    ; Control flow
    string-map vector-map string-for-each vector-for-each
    ; Exception
    raise guard read-error? file-error?)
  (begin
```

goldfish/liii/base.scm

△ 2 ▽

```

(define-library (liii base)
  (import (scheme base)
          (srfi srfi-2)
          (srfi srfi-8))
  (export
    ; (scheme base) defined by R7RS
    let-values
    define-record-type
    square
    exact inexact
    floor s7-floor ceiling s7-ceiling truncate s7-truncate round s7-round
    floor-quotient
    gcd lcm s7-lcm
    boolean=?
    ; String
    string-copy
    ; Vector
    vector->string string->vector
    vector-copy vector-copy! vector-fill!
    ; Input and Output
    call-with-port port? binary-port? textual-port?
    input-port-open? output-port-open?
    open-binary-input-file open-binary-output-file
    close-port
    eof-object
    ; Control flow
    string-map vector-map string-for-each vector-for-each
    ; Exception
    raise guard read-error? file-error?
    ; SRFI-2
    and-let*
    ; SRFI-8
    receive
    ; Extra routines for (liii base)
    == != display* in? let1 compose identity typed-lambda
  )
  (begin

```

3.3 测试

tests/goldfish/liii/base-test.scm

△ 2 ▽

```

(import (liii check)
        (liii base)
        (liii list))

(check-set-mode! 'report-failed)

```

3.4 表达式

本节对应R7RS的第四节：表达式。

3.4.1 原语表达式

3.4.2 派生表达式

cond

 
case

tests/goldfish/liii/base-test.scm

△ 3 ▽

```
(check (case '+
            ((+ -) 'p0)
            ((* /) 'p1))
=> 'p0)

(check (case '-
            ((+ -) 'p0)
            ((* /) 'p1))
=> 'p0)

(check (case '*
            ((+ -) 'p0)
            ((* /) 'p1))
=> 'p1)

(check (case '@
            ((+ -) 'p0)
            ((* /) 'p1))
=> #<unspecified>)

(check (case '&
            ((+ -) 'p0)
            ((* /) 'p1))
=> #<unspecified>)
```

 
and

检查 **and** 是否正确处理多个布尔表达式。

tests/goldfish/liii/base-test.scm

△ 4 ▽

```
(check-true (and #t #t #t))
(check-false (and #t #f #t))
(check-false (and #f #t #f))
(check-false (and #f #f #f))
```

验证当 **and** 没有参数时的行为。

tests/goldfish/liii/base-test.scm

△ 5 ▽

```
(check-true (and))
```

测试 **and** 与混合类型参数的组合。

tests/goldfish/liii/base-test.scm△ 6 ▾

```
(check-true (and 1 '() "non-empty" #t))
(check-false (and #f '() "non-empty" #t))
(check-false (and 1 '() "non-empty" #f))
```

检查 and 在复合表达式中的行为。

tests/goldfish/liii/base-test.scm△ 7 ▾

```
(check-true (and (> 5 3) (< 5 10)))
(check-false (and (> 5 3) (> 5 10)))
```

验证 and 的短路行为。

tests/goldfish/liii/base-test.scm△ 8 ▾

```
(check-catch 'error-name
  (and (error 'error-name "This should not be evaluated") #f))
(check-false (and #f (error "This should not be evaluated")))
```

验证and返回值非布尔值的情况。

tests/goldfish/liii/base-test.scm△ 9 ▾

```
(check (and #t 1) => 1)
```

✚ 7 行

索引

or

✚ 7 行

when

✚ 7 行

unless

✚ 7 行

索引

let

✚ 7 行

let*

✚ 7 行

letrec

tests/goldfish/liii/base-test.scm△ 10 ▾

```
(define (test-letrec)
  (letrec ((even?
    (lambda (n)
      (if (= n 0)
        #t
        (odd? (- n 1))))))
    (odd?
      (lambda (n)
        (if (= n 0)
          #f
          (even? (- n 1))))))
    (list (even? 10) (odd? 10))))

(check (test-letrec) => (list #t #f))

(check-catch 'wrong-type-arg
  (letrec ((a 1) (b (+ a 1))) (list a b)))
```

✚ 7 行

索引

letrec*

tests/goldfish/liii/base-test.scm

△ 11 ▾

```
(check
  (letrec* ((a 1) (b (+ a 1))) (list a b))
  => (list 1 2))
```

let-values

索引

goldfish/scheme/base.scm

△ 3 ▾

```
; 0-clause BSD
; Bill Schottstaedt
; from S7 source repo: r7rs.scm
(define-macro (let-values vars . body)
  (if (and (pair? vars)
           (pair? (car vars))
           (null? (cdar vars)))
      '((lambda ,(caar vars)
           ,@body)
        ,(cadar vars))
      '(with-let
         (apply sublet (curlet)
                  (list
                     ,@(map
                        (lambda (v)
                          '((lambda ,(car v)
                             (values ,@(map (lambda (name)
                                                (values (symbol->keyword name) name))
                                                (let args->proper-list ((args (car v)))
                                                  (cond ((symbol? args)
                                                         (list args))
                                                         ((not (pair? args))
                                                          args)
                                                         ((pair? (car args))
                                                         (cons (caar args)
                                                                (args->proper-list (cdr args))))
                                                         (else
                                                          (cons (car args)
                                                                (args->proper-list (cdr args)))))))
                             , (cadr v)))
                          vars)))
                     ,@body)))
```

tests/goldfish/liii/base-test.scm

△ 12 ▾

```
(check (let-values (((ret) (+ 1 2))) (+ ret 4)) => 7)
(check (let-values (((a b) (values 3 4))) (+ a b)) => 7)
```

let*-values

and-let*

索引

tests/goldfish/liii/base-test.scm

△ 13 ▾

```
(check (and-let* ((hi 3) (ho #f)) (+ hi 1)) => #f)
(check (and-let* ((hi 3) (ho #t)) (+ hi 1)) => 4)
```

goldfish/srfi/srfi-2.scm

```
; 0-clause BSD by Bill Schottstaedt from S7 source repo: s7test.scm
(define-library (srfi srfi-2)
  (export and-let*)
  (begin

(define-macro (and-let* vars . body)
  '(let () (and ,@map (lambda (v) '(define ,@v)) vars) (begin ,@body))))

) ; end of begin
) ; end of define-library
```

3.5 程序结构

本节对应R7RS的第5节：程序结构。

define-record-type

[索引](#)

实现

goldfish/scheme/base.scm

[△](#) [4](#) [▽](#)

```
; 0-clause BSD by Bill Schottstaedt from S7 source repo: r7rs.scm
(define-macro (define-record-type type make ? . fields)
  (let ((obj (gensym))
        (typ (gensym)) ; this means each call on this macro makes a new type
        (args (map (lambda (field)
                      (values (list 'quote (car field))
                                (let ((par (memq (car field) (cdr make))))
                                  (and (pair? par) (car par))))))
                    fields)))
    '(begin
      (define (,? ,obj)
        (and (let? ,obj)
              (eq? (let-ref ,obj ',typ) ',type)))

      (define ,make
        (inlet ',typ ',type ,@args))

      ,@map
        (lambda (field)
          (when (pair? field)
            (if (null? (cdr field))
                (values)
                (if (null? (cddr field))
                    '(define (, (cadr field) ,obj)
                      (let-ref ,obj ', (car field)))
                    '(begin
                      (define (, (cadr field) ,obj)
                        (let-ref ,obj ', (car field)))
                      (define (, (caddr field) ,obj val)
                        (let-set! ,obj ', (car field) val))))))
            fields)
      ',type)))
```

测试

通过`define-record-type`，定义了一种名为`pare`的记录类型，其中`kons`是这种记录类型的构造器，`pare?`是谓词，`kar`和`kdr`是选择器，`set-kar!`是修改器。

tests/goldfish/liii/base-test.scm

△ 14 ▽

```
(define-record-type :pare
  (kons x y)
  pare?
  (x kar set-kar!)
  (y kdr))

(check (pare? (kons 1 2)) => #t)
(check (pare? (cons 1 2)) => #f)
(check (kar (kons 1 2)) => 1)
(check (kdr (kons 1 2)) => 2)

(check
 (let ((k (kons 1 2)))
   (set-kar! k 3)
   (kar k))
 => 3)
```

上面那个例子比较难懂，提供一个更易懂的例子：

tests/goldfish/liii/base-test.scm

△ 15 ▽

```
(define-record-type :person
  (make-person name age)
  person?
  (name get-name set-name!)
  (age get-age))

(check (person? (make-person "Da" 3)) => #t)
(check (get-age (make-person "Da" 3)) => 3)
(check (get-name (make-person "Da" 3)) => "Da")
(check
 (let ((da (make-person "Da" 3)))
   (set-name! da "Darcy")
   (get-name da))
 => "Darcy")
```

SRFI

goldfish/srfi/srfi-9.scm

```
(define-library (srfi srfi-9)
  (import (scheme base))
  (export define-record-type)
  (begin
   ) ; end of begin
  ) ; end of define-library
```

3.6 相等性判断

3.7 数

本节对应R7RS的6.2节。

3.7.1 数的类型

索引

number? (number? obj) => boolean

判断一个对象是否是数。

complex?

判断一个对象是否是复数。

real?

rational?

integer?

3.7.2 数的准确性

索引

exact?

[#e3.0需要支持一下]

tests/goldfish/liii/base-test.scm

△ 16 ▾

(check-true (exact? 1))
(check-true (exact? 1/2))
(check-false (exact? 0.3))
; (check-true (exact? #e3.0))

inexact?

exact-integer?

exact

goldfish/scheme/base.scm

△ 5 ▾

(define exact inexact->exact)

inexact

goldfish/scheme/base.scm

△ 6 ▾

(define inexact exact->inexact)

3.7.3 数的运算

zero?

索引

floor (x) => integer

返回最大的不大于 x 的整数。如果 x 是准确值，那么返回值也是准确值，如果 x 不是准确值，那么返回值也不是准确值。

goldfish/scheme/base.scm

△ 7 ▾

```
(define s7-floor floor)

(define (floor x)
  (if (inexact? x)
      (inexact (s7-floor x))
      (s7-floor x)))
```

tests/goldfish/liii/base-test.scm

△ 17 ▾

```
(check (floor 1.1) => 1.0)
(check (floor 1) => 1)
(check (floor 1/2) => 0)
(check (floor 0) => 0)
(check (floor -1) => -1)
(check (floor -1.2) => -2.0)

(check (s7-floor 1.1) => 1)
(check (s7-floor -1.2) => -2)
```

ceiling

goldfish/scheme/base.scm

△ 8 ▾

```
(define s7-ceiling ceiling)

(define (ceiling x)
  (if (inexact? x)
      (inexact (s7-ceiling x))
      (s7-ceiling x)))
```

tests/goldfish/liii/base-test.scm

△ 18 ▾

```
(check (ceiling 1.1) => 2.0)
(check (ceiling 1) => 1)
(check (ceiling 1/2) => 1)
(check (ceiling 0) => 0)
(check (ceiling -1) => -1)
(check (ceiling -1.2) => -1.0)

(check (s7-ceiling 1.1) => 2)
(check (s7-ceiling -1.2) => -1)
```

truncate

goldfish/scheme/base.scm

△ 9 ▾

```
(define s7-truncate truncate)

(define (truncate x)
  (if (inexact? x)
      (inexact (s7-truncate x))
      (s7-truncate x)))
```

tests/goldfish/liii/base-test.scm

△ 19 ▽

```
(check (truncate 1.1) => 1.0)
(check (truncate 1) => 1)
(check (truncate 1/2) => 0)
(check (truncate 0) => 0)
(check (truncate -1) => -1)
(check (truncate -1.2) => -1.0)

(check (s7-truncate 1.1) => 1)
(check (s7-truncate -1.2) => -1)
```

round

goldfish/scheme/base.scm

△ 10 ▽

```
(define s7-round round)

(define (round x)
  (if (inexact? x)
      (inexact (s7-round x))
      (s7-round x)))
```

tests/goldfish/liii/base-test.scm

△ 20 ▽

```
(check (round 1.1) => 1.0)
(check (round 1.5) => 2.0)
(check (round 1) => 1)
(check (round 1/2) => 0)
(check (round 0) => 0)
(check (round -1) => -1)
(check (round -1.2) => -1.0)
(check (round -1.5) => -2.0)
```

x7718

索引

floor-quotient `((x real?) (y real?)) => integer`

$$\text{floor_quotient}(y, x) = \lfloor y/x \rfloor$$

goldfish/scheme/base.scm

△ 11 ▽

```
(define (floor-quotient x y) (floor (/ x y)))
```

测试

正常情况测试

tests/goldfish/liii/base-test.scm

△ 21 ▽

```
(check (floor-quotient 11 2) => 5)
(check (floor-quotient 11 -2) => -6)
(check (floor-quotient -11 2) => -6)
(check (floor-quotient -11 -2) => 5)

(check (floor-quotient 10 2) => 5)
(check (floor-quotient 10 -2) => -5)
(check (floor-quotient -10 2) => -5)
(check (floor-quotient -10 -2) => 5)
```

特殊情况测试

tests/goldfish/liii/base-test.scm

△ 22 ▽

```
(check-catch 'division-by-zero (floor-quotient 11 0))
(check-catch 'division-by-zero (floor-quotient 0 0))

(check (floor-quotient 0 2) => 0)
(check (floor-quotient 0 -2) => 0)
```

x7rs

索引

quotient `(quotient (y real?) (x real?)) => integer`

求两个数的商。

$$\text{quotient}(y, x) = \text{truncate}(y/x)$$

tests/goldfish/liii/base-test.scm

△ 23 ▽

```
(check (quotient 11 2) => 5)
(check (quotient 11 -2) => -5)
(check (quotient -11 2) => -5)
(check (quotient -11 -2) => 5)

(check-catch 'division-by-zero (quotient 11 0))
(check-catch 'division-by-zero (quotient 0 0))
(check-catch 'wrong-type-arg (quotient 1+i 2))
```

remainder

modulo

x7rs

索引

gcd `(x ...) => (y positive?)`

求n个数的最大公约数。

tests/goldfish/liii/base-test.scm

△ 24 ▽

```
(check (gcd) => 0)
(check (gcd 0) => 0)
(check (gcd 1) => 1)
(check (gcd 2) => 2)
(check (gcd -1) => 1)

(check (gcd 0 1) => 1)
(check (gcd 1 0) => 1)
(check (gcd 1 2) => 1)
(check (gcd 1 10) => 1)
(check (gcd 2 10) => 2)
(check (gcd -2 10) => 2)

(check (gcd 2 3 4) => 1)
(check (gcd 2 4 8) => 2)
(check (gcd -2 4 8) => 2)
```

lcm

```
(define s7-lcm lcm)

(define (lcm2 x y)
  (cond ((and (inexact? x) (exact? y))
         (inexact (s7-lcm (exact x) y)))
        ((and (exact? x) (inexact? y))
         (inexact (s7-lcm x (exact y))))
        ((and (inexact? x) (inexact? y))
         (inexact (s7-lcm (exact x) (exact y))))
        (else (s7-lcm x y))))

(define (lcm . args)
  (cond ((null? args) 1)
        ((null? (cdr args))
         (car args))
        ((null? (cddr args))
         (lcm2 (car args) (cadr args)))
        (else (apply lcm (cons (lcm (car args) (cadr args))
                                (cddr args))))))
```

```
(check (lcm) => 1)
(check (lcm 1) => 1)
(check (lcm 0) => 0)
(check (lcm 32 -36) => 288)
(check (lcm 32 -36.0) => 288.0)
(check (lcm 2 4) => 4)
(check (lcm 2 4.0) => 4.0)
(check (lcm 2.0 4.0) => 4.0)
(check (lcm 2.0 4) => 4.0)
```

平方

索引

square

求一个数的平方。

```
(define (square x) (* x x))
```

```
(check (square 2) => 4)
```

sqrt

exact-integer-sqrt

expt

3.7.4 数的转换

3.8 布尔值

布尔值

索引

boolean=? (obj1 obj2 ...) => boolean

goldfish/scheme/base.scm△ 14 ▾

```
(define (boolean=? obj1 obj2 . rest)
  (define (same-boolean obj rest)
    (if (null? rest)
        #t
        (and (equal? obj (car rest))
              (same-boolean obj (cdr rest))))))
(cond ((not (boolean? obj1)) #f)
      ((not (boolean? obj2)) #f)
      ((not (equal? obj1 obj2)) #f)
      (else (same-boolean obj1 rest))))
```

测试

tests/goldfish/liii/base-test.scm△ 27 ▾

```
(check-true (boolean=? #t #t))
(check-true (boolean=? #f #f))
(check-true (boolean=? #t #t #t))
(check-false (boolean=? #t #f))
(check-false (boolean=? #f #t))
```

3.9 序对和列表

见章 8

3.10 字符

char? 索引
判断一个对象x是否是字符类型。

tests/goldfish/liii/base-test.scm△ 28 ▾

```
(check (char? #\A) => #t)
(check (char? 1) => #f)
```

char=? 索引
判断两个及以上字符对象是否相等。

tests/goldfish/liii/base-test.scm△ 29 ▾

```
(check (char=? #\A #\A) => #t)
(check (char=? #\A #\A #\A) => #t)
(check (char=? #\A #\a) => #f)
```

3.11 字符串

见章 7

3.12 向量

见章 9

3.13 字节向量

3.14 控制流

procedure? 索引

apply 索引

apply是R7RS定义的函数。

```
tests/goldfish/liii/base-test.scm △ 30 ▽
(check (apply + (list 3 4)) => 7)
(check (apply + (list 2 3 4)) => 9)
```

在这个例子中，(apply + (list 3 4))实际被展开为(+ 3 4)。+这个函数接受两个参数，但是无法接受一个列表，利用apply就可以把列表展开并作为+的两个参数。

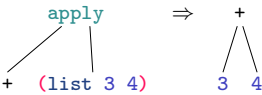


图 3.1. apply的原理可视化

call-with-current-continuation

call/cc

values

R7RS
call-with-values (producer consumer)

```
tests/goldfish/liii/base-test.scm △ 31 ▽
(check (call-with-values (lambda () (values 4 5))
                          (lambda (x y) x))
      => 4)
```

由于*这个函数在没有参数时返回1，而-在只有一个参数的时候，返回该值的相反数。故而下面这个测试用例的返回值是-1。

```
tests/goldfish/liii/base-test.scm △ 32 ▽
(check (*) => 1)
(check (call-with-values * -) => -1)
```

receive 索引

官网：<https://srfi.schemers.org/srfi-8/srfi-8.html>

官网的参考实现是：

```
(define-syntax receive
```

```
(syntax-rules ()
  ((receive formals expression body ...)
   (call-with-values (lambda () expression)
                     (lambda formals body ...))))
```

但是S7 Scheme里面没有define-syntax和syntax-rule，在墨干中的实现是这样的：

```
> (define-macro (receive vars vals . body)
  '((lambda ,vars ,@body) ,vals)) ; GPL
> (receive (a b) (values 1 2) (+ a b))
> ((lambda (a b) (+ a b)) 1 2)
>
```

S7里面有call-with-values，使用define-macro就可以实现SRFI-8。

goldfish/srfi/srfi-8.scm

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;

(define-library (srfi srfi-8)
  (export receive)
  (begin

    (define-macro (receive formals expression . body)
      '(call-with-values
        (lambda () (values ,expression))
        (lambda ,formals ,@body)))

    ) ; end of begin
  ) ; end of define-library
```


tests/goldfish/liii/base-test.scm

△ 33 ▽

```
(check
  (receive (a b) (values 1 2) (+ a b))
  => 3)
```

dynamic-wind

3.15 异常处理

guard 

goldfish/scheme/base.scm

△ 15 ▽

```

(define (raise . args)
  (apply throw #t args))

(define-macro (guard results . body)
  '(let ((,(car results)
        (catch #t
          (lambda ()
            ,@body)
          (lambda (type info)
            (if (pair? (*s7* 'catches))
                (lambda () (apply throw type info))
                (car info))))))
    (cond ,@(cdr results)
      (else
       (if (procedure? ,(car results))
           ,(car results)
           ,(car results))))))

```

测试用例

tests/goldfish/liii/base-test.scm

△ 34 ▽

```

(guard (condition
  (else
    (display "condition: ")
    (write condition)
    (newline)
    'exception))
  (+ 1 (raise 'an-error)))
; PRINTS: condition: an-error

(guard (condition
  (else
    (display "something went wrong")
    (newline)
    'dont-care))
  (+ 1 (raise 'an-error)))
; PRINTS: something went wrong

```

read-error?

索引

goldfish/scheme/base.scm

△ 16 ▽

```

(define (read-error? obj) (eq? (car obj) 'read-error))

```

file-error?

索引

goldfish/scheme/base.scm

△ 17 ▽

```

(define (file-error? obj) (eq? (car obj) 'io-error))

```

3.16 输入和输出

3.16.1 端口

call-with-port

索引

goldfish/scheme/base.scm

△ 18 ▾

```
(define (call-with-port port proc)
  (let ((res (proc port)))
    (if res (close-port port))
    res)))
```

input-port?

索引

S7内置函数。

output-port?

索引

S7内置函数。

port?

索引

goldfish/scheme/base.scm

△ 19 ▾

```
(define (port? p) (or (input-port? p) (output-port? p)))
```

textual-port?

索引

goldfish/scheme/base.scm

△ 20 ▾

```
(define textual-port? port?)
```

binary-port?

索引

goldfish/scheme/base.scm

△ 21 ▾

```
(define binary-port? port?)
```

input-port-open?

索引

goldfish/scheme/base.scm

△ 22 ▾

```
(define (input-port-open? p) (not (port-closed? p)))
```

output-port-open?

索引

goldfish/scheme/base.scm

△ 23 ▾

```
(define (output-port-open? p) (not (port-closed? p)))
```

close-port

索引

goldfish/scheme/base.scm

△ 24 ▾

```
(define (close-port p)
  (if (input-port? p)
      (close-input-port p)
      (close-output-port p)))
```

close-input-port

索引

S7内置函数。

close-output-port

索引

S7内置函数。

3.16.2 输入

read

索引

tests/goldfish/liii/base-test.scm

△ 35 ▾

```
(with-input-from-string "(+ 1 2)"
  (lambda ()
    (let ((datum (read)))
      (check-true (list? datum))
      (check datum => '(+ 1 2))))
```

read-char

peek-char

read-line

eof-object?

eof-object

索引

实现

goldfish/scheme/base.scm

△ 25 ▾

```
(define (eof-object) #<eof>)
```

测试

tests/goldfish/liii/base-test.scm

△ 36 ▾

```
(check (eof-object) => #<eof>)
```

char-ready?

read-string

read-u8

peek-u8

u8-ready?

read-bytevector

read-bytevector!

3.16.3 输出

3.17 系统接口

3.18 三鲤扩展库

== 索引 (x y) => bool
equal?的语法糖。

goldfish/liii/base.scm

△ 3 ▾

```
(define == equal?)
```

tests/goldfish/liii/base-test.scm

△ 37 ▾

```
(check (== (list 1 2) (list 1 2)) => #t)
(check (!= (list 1 2) (list 1 2)) => #f)
```

 `!= (x y) => bool`

语法糖。

goldfish/liii/base.scm

△ 4 ▽

```
(define (!= left right)
  (not (equal? left right)))
```

tests/goldfish/liii/base-test.scm

△ 38 ▽

```
(check (== (list 1 2) (list 1 2)) => #t)
(check (!= (list 1 2) (list 1 2)) => #f)
```

 `display* (x y z ...) => <#unspecified>`

`display*`可以输入多个参数，是`display`的加强版。

goldfish/liii/base.scm


△ 5 ▽

```
(define (display* . params)
  (for-each display params))
```

tests/goldfish/liii/base-test.scm

△ 39 ▽

```
(check
  (with-output-to-string
    (lambda ()
      (display* "hello world" "\n"))))
=> "hello world\n")
```

 `in? (x sequence) => bool`

判断一个元素是否在对象中：

1. 一个元素是否在列表中
2. 一个元素是否在向量中
3. 一个字符是否在字符串中

其中的相等性判断使用的是`==`，也就是R7RS定义的`equal?`。

tests/goldfish/liii/base-test.scm

△ 40 ▽

```
(check (in? 1 (list )) => #f)
(check (in? 1 (list 3 2 1)) => #t)
(check (in? #\x "texmacs") => #t)
(check (in? 1 (vector )) => #f)
(check (in? 1 (vector 3 2 1)) => #t)
(check-catch 'type-error (in? 1 "123"))
```

goldfish/liii/base.scm

△ 6 ▽

```
(define (in? elem l)
  (cond ((list? l) (not (not (member elem l))))
        ((vector? l)
         (let loop ((i (- (vector-length l) 1)))
           (if (< i 0)
               #f
               (if (== elem (vector-ref l i))
                   #t
                   (loop (- i 1))))))
        ((and (char? elem) (string? l))
         (in? elem (string->list l)))
        (else (error 'type-error "type mismatch"))))
```

let1

let的语法嵌套层次太多了，故而引入let1，作为let的单参数版本，简化语法。

[goldfish/liii/base.scm](#)
[△ 7 ▽](#)

```
(define-macro (let1 name1 value1 . body)
  `(let ((,name1 ,value1))
    ,@body))
```

[tests/goldfish/liii/base-test.scm](#)
[△ 41 ▽](#)

```
(check (let1 x 1 x) => 1)
(check (let1 x 1 (+ x 1)) => 2)
```

identity

[索引](#) (x) => x

[goldfish/liii/base.scm](#)
[△ 8 ▽](#)

```
(define identity (lambda (x) x))
```

compose

[索引](#) (f g ...) => f

[goldfish/liii/base.scm](#)
[△ 9 ▽](#)

```
(define (compose . fs)
  (if (null? fs)
      (lambda (x) x)
      (lambda (x)
        ((car fs) ((apply compose (cdr fs)) x))))))
```

[tests/goldfish/liii/base-test.scm](#)
[△ 42 ▽](#)

```
(check-true ((compose not zero?) 1))
(check-false ((compose not zero?) 0))
```

typed-lambda

[索引](#)

实现
[goldfish/liii/base.scm](#)
[△ 10 ▽](#)

```
; 0 clause BSD, from S7 repo stuff.scm
(define-macro (typed-lambda args . body)
  ; (typed-lambda ((var [type])...) ...)
  (if (symbol? args)
      (apply lambda args body)
      (let ((new-args (copy args)))
        (do ((p new-args (cdr p)))
            ((not (pair? p)))
            (if (pair? (car p))
                (set-car! p (caar p))))
        `(lambda ,new-args
          ,@(map (lambda (arg)
                    (if (pair? arg)
                        `(unless (,(cadr arg) ,(car arg))
                          (error 'type-error
                                "~S is not ~S~%" ',(car arg) ',(cadr arg)))
                        (values)))
                  args)
          ,@body))))))
```

测试

`tests/goldfish/liii/base-test.scm`[△ 43](#) [▽](#)

```
(define add3
  (typed-lambda
    ((i integer?) (x real?) z)
    (+ i x z)))

(check (add3 1 2 3) => 6)
(check-catch 'type-error (add3 1.2 2 3))
```

3.19 结尾

`goldfish/liii/base.scm`[△ 11](#)

```
) ; end of begin
) ; end of define-library
```

`tests/goldfish/liii/base-test.scm`[△ 44](#) [▽](#)

```
(check-report)
```

第 4 章

(liii error)

异常的命名参考Python标准库的[内置异常](#)。

4.1 许可证

[goldfish/liii/error.scm](#)

1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

[tests/goldfish/liii/error-test.scm](#)

1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

4.2 接口

goldfish/liii/error.scm△ 2 ▽

```
; see https://docs.python.org/3/library/exceptions.html#exception-hierarchy
(define-library (liii error)
  (export ???
    os-error file-not-found-error not-a-directory-error file-exists-error
    timeout-error
    type-error type-error? value-error)
  (begin
```

4.3 测试

tests/goldfish/liii/error-test.scm△ 2 ▽

```
(import (liii check)
        (liii error)
        (liii base))

(check-set-mode! 'report-failed)
```

4.4 实现

os-error 

系统级别的错误。

goldfish/liii/error.scm△ 3 ▽

```
(define (os-error . args)
  (apply error (cons 'os-error args)))
```

tests/goldfish/liii/error-test.scm△ 3 ▽

```
(check-catch 'os-error (os-error))
```

file-not-found-error 

文件未找到。

goldfish/liii/error.scm△ 4 ▽

```
(define (file-not-found-error . args)
  (apply error (cons 'file-not-found-error args)))
```

tests/goldfish/liii/error-test.scm△ 4 ▽

```
(check-catch 'file-not-found-error (file-not-found-error))
```

not-a-directory-error 

不是一个目录。

goldfish/liii/error.scm△ 5 ▽

```
(define (not-a-directory-error . args)
  (apply error (cons 'not-a-directory-error args)))
```

tests/goldfish/liii/error-test.scm

△ 5 ▾

```
(check-catch 'not-a-directory-error (not-a-directory-error))
```

file-exists-error

索引

文件已存在。

goldfish/liii/error.scm

△ 6 ▾

```
(define (file-exists-error . args)
  (apply error (cons 'file-exists-error args)))
```

tests/goldfish/liii/error-test.scm

△ 6 ▾

```
(check-catch 'file-exists-error (file-exists-error))
```

timeout-error

索引

超时错误。

goldfish/liii/error.scm

△ 7 ▾

```
(define (timeout-error . args)
  (apply error (cons 'timeout-error args)))
```

tests/goldfish/liii/error-test.scm

△ 7 ▾

```
(check-catch 'timeout-error (timeout-error))
```

type-error

索引

如果类型不匹配，直接报错。

goldfish/liii/error.scm

△ 8 ▾

```
(define (type-error . args)
  (apply error (cons 'type-error args)))
```

tests/goldfish/liii/error-test.scm

△ 8 ▾

```
(check-catch 'type-error (type-error))
(check-catch 'type-error (type-error "msg"))
(check-catch 'type-error (type-error "msg" "msg2"))
```

type-error?

索引

goldfish/liii/error.scm

△ 9 ▾

```
(define (type-error? err)
  (in? err '(type-error wrong-type-arg)))
```

tests/goldfish/liii/error-test.scm

△ 9 ▾

```
(check-true (type-error? 'type-error))
(check-true (type-error? 'wrong-type-arg))
```

value-error

索引

值不正确

goldfish/liii/error.scm

△ 10 ▾

```
(define (value-error . args)
  (apply error (cons 'value-error args)))
```

tests/goldfish/liii/error-test.scm

△ 10 ▾

(check-catch 'value-error (value-error))

???

Scala风格未实现错误，一般用于标记为实现的接口。

goldfish/liii/error.scm

△ 11 ▾

(define (???)
 (error 'not-implemented-error "???"))

tests/goldfish/liii/error-test.scm

△ 11 ▾

(check-catch 'not-implemented-error (???))

4.5 结尾

goldfish/liii/error.scm

△ 12

) ; begin
) ; define-library

tests/goldfish/liii/error-test.scm

△ 12

(check-report)

第 5 章

(liii check)

5.1 协议

[goldfish/liii/check.scm](#)

1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

```
; <PLAINTEXT>
; Copyright (c) 2005-2006 Sebastian Egner.
;
; Permission is hereby granted, free of charge, to any person obtaining
; a copy of this software and associated documentation files (the
; ''Software''), to deal in the Software without restriction, including
; without limitation the rights to use, copy, modify, merge, publish,
; distribute, sublicense, and/or sell copies of the Software, and to
; permit persons to whom the Software is furnished to do so, subject to
; the following conditions:
;
; The above copyright notice and this permission notice shall be
; included in all copies or substantial portions of the Software.
;
; THE SOFTWARE IS PROVIDED ''AS IS'', WITHOUT WARRANTY OF ANY KIND,
; EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
; MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
; NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
; LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
; OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
; WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
;
; -----
; Lightweight testing (reference implementation)
; =====
;
; Sebastian.Egner@philips.com
; in R5RS + SRFI 23 (error) + SRFI 42 (comprehensions)
;
; history of this file:
;   SE, 25-Oct-2004: first version based on code used in SRFIs 42 and 67
;   SE, 19-Jan-2006: (arg ...) made optional in check-ec
;
; Naming convention "check:<identifier>" is used only internally.
;
; Copyright (c) 2024 The Goldfish Scheme Authors
; Follow the same License as the original one
```

tests/goldfish/srfi/srfi-78-test.scm1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

5.2 接口

将check:proc导出的原因是check是使用宏实现的，而宏里面用到了check:proc，不导出无法生效。

goldfish/liii/check.scm△ 2 ▾

```
(define-library (liii check)
  (export test check check-set-mode! check:proc
    check-catch check-report check-failed?
    check-true check-false)
  (import (srfi srfi-78)
    (rename (srfi srfi-78)
      (check-report srfi-78-check-report)))
  (begin
```

goldfish/srfi/srfi-78.scm△ 2 ▾

```
(define-library (srfi srfi-78)
  (export check check-set-mode! check-report check-reset!
    check-passed? check-failed?
    check:proc)
  (begin
```

5.3 实现

使用display作为测试结果的展示函数，比write好，因为display可以正常显示文本中的汉字。

goldfish/srfi/srfi-78.scm△ 3 ▾

```
(define check:write display)
```

check-set-mode!索引

goldfish/srfi/srfi-78.scm

△ 4 ▾

```
(define check:mode #f)

(define (check-set-mode! mode)
  (set! check:mode
    (case mode
      ((off) 0)
      ((summary) 1)
      ((report-failed) 10)
      ((report) 100)
      (else (error "unrecognized mode" mode))))))
```

将检查模式初始化为 `'report`:

goldfish/srfi/srfi-78.scm

△ 5 ▾

```
(check-set-mode! 'report)
```

以 `check` 函数为例，不同的检查模式下，得到的结果不同。可以使用单元测试来查看这四种测试用例具体使用方法：

tests/goldfish/srfi/srfi-78-test.scm

△ 2 ▾

```
(import (srfi srfi-78))
```

report. 默认检查模式，无论正确还是错误，都会展示详细的信息。

tests/goldfish/srfi/srfi-78-test.scm

△ 3 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;

(display "-----\n")
(display "check mode: report\n")

(check-set-mode! 'report)

(check (+ 1 2) => 3)

(check (+ 1 2) => 4)

(check-reset!)
```

off. 设置为off的时候，禁用后续的测试用例。

```
tests/goldfish/srfi/srfi-78-test.scm
```

```
△ 4 ▽
```

```
(display "\n-----\n")
(display "check mode: off\n")

(check-set-mode! 'off)

(check (+ 1 2) => 3)

(check (+ 1 2) => 4)

(check-reset!)
```

report-failed. 设置为report-failed的时候，在check正确时只返回正确的测试数量，在check错误时，显示错误。

```
tests/goldfish/srfi/srfi-78-test.scm
```

```
△ 5 ▽
```

```
(display "\n-----\n")
(display "check mode: report-failed\n")

(check-set-mode! 'report-failed)

(check (+ 1 2) => 3)

(check (+ 1 2) => 4)

(check-reset!)
```

summary. 设置为summary的时候，不汇报错误，需要显示调用**check-report**才能显示的测试汇总结果，显式调用**check:failed**显式查看错误的例子。

```
tests/goldfish/srfi/srfi-78-test.scm
```

```
△ 6 ▽
```

```
(display "\n-----\n")
(display "check mode: summary\n")

(check-set-mode! 'summary)

(check (+ 1 2) => 3)

(check (+ 1 2) => 4)

(check-report)
```

这个时候如果需要查看错误，那么可以重新设置检查模式，重新检查报告：

```
tests/goldfish/srfi/srfi-78-test.scm
```

```
△ 7
```

```
(check-set-mode! 'report)

(check-report)
```

goldfish/srfi/srfi-78.scm

△ 6 ▽

```
(define check:correct 0)
(define check:failed '())

(define (check-reset!)
  (set! check:correct 0)
  (set! check:failed '()))

(define (check:add-correct!)
  (set! check:correct (+ check:correct 1)))

(define (check:add-failed! expression actual-result expected-result)
  (set! check:failed
    (cons (list expression actual-result expected-result)
          check:failed)))
```

goldfish/srfi/srfi-78.scm

△ 7 ▽

```
(define (check:report-expression expression)
  (newline)
  (check:write expression)
  (display " => "))

(define (check:report-actual-result actual-result)
  (check:write actual-result)
  (display " ; "))

(define (check:report-correct cases)
  (display "correct")
  (if (not (= cases 1))
      (begin (display " (")
              (display cases)
              (display " cases checked"))))
  (newline))

(define (check:report-failed expected-result)
  (display "*** failed ***")
  (newline)
  (display "; expected result: ")
  (check:write expected-result)
  (newline))

(define (check-passed? expected-total-count)
  (and (= (length check:failed) 0)
       (= check:correct expected-total-count)))

(define (check-failed?)
  (>= (length check:failed) 1))
```

check

索引

check的具体过程，依据不同的检查模式，做不同的处理：

goldfish/srfi/srfi-78.scm

△ 8 ▽

```
(define (check:proc expression thunk equal expected-result)
  (case check:mode
    ((0) #f)
    (1)
      (let ((actual-result (thunk)))
        (if (equal actual-result expected-result)
            (check:add-correct!)
            (check:add-failed! expression actual-result expected-result))))
    (10)
      (let ((actual-result (thunk)))
        (if (equal actual-result expected-result)
            (check:add-correct!)
            (begin
              (check:report-expression expression)
              (check:report-actual-result actual-result)
              (check:report-failed expected-result)
              (check:add-failed! expression actual-result expected-result))))))
    (100)
      (check:report-expression expression)
      (let ((actual-result (thunk)))
        (check:report-actual-result actual-result)
        (if (equal actual-result expected-result)
            (begin (check:report-correct 1)
                  (check:add-correct!))
            (begin (check:report-failed expected-result)
                  (check:add-failed! expression
                                     actual-result
                                     expected-result))))))
    (else (error "unrecognized check:mode" check:mode))))
```

check的接口实现，使用S7 Scheme的define-macro实现：

goldfish/srfi/srfi-78.scm

△ 9 ▽

```
(define-macro (check expr => expected)
  `(check:proc ',expr (lambda () ,expr) equal? ,expected))
```

check-true

索引

goldfish/liii/check.scm

△ 3 ▽

```
(define-macro (check-true body)
  `(check ,body => #t))
```

check-false

索引

goldfish/liii/check.scm

△ 4 ▽

```
(define-macro (check-false body)
  `(check ,body => #f))
```

check-catch

索引

goldfish/liii/check.scm

△ 5 ▽

```
(define-macro (check-catch error-id body)
  '(check
    (catch ,error-id
      (lambda () ,body)
      (lambda args ,error-id))
    => ,error-id))
```

test (obj1 obj2) => boolean

test函数是为了兼容S7 Scheme仓库里面的测试用例。

goldfish/liii/check.scm

△ 6 ▽

```
(define-macro (test left right)
  '(check ,left => ,right))
```

check-report

索引

goldfish/liii/check.scm

△ 7 ▽

```
(define (check-report . msg)
  (if (not (null? msg))
      (begin
        (display (car msg)))
      (srfi-78-check-report)
      (if (check-failed?) (exit -1))))
```

goldfish/srfi/srfi-78.scm

△ 10 ▽

```
(define (check-report)
  (if (>= check:mode 1)
      (begin
        (newline)
        (display "; *** checks *** : ")
        (display check:correct)
        (display " correct, ")
        (display (length check:failed))
        (display " failed.")
        (if (or (null? check:failed) (<= check:mode 1))
            (newline)
            (let* ((w (car (reverse check:failed)))
                   (expression (car w))
                   (actual-result (cadr w))
                   (expected-result (caddr w)))
              (display " First failed example:")
              (newline)
              (check:report-expression expression)
              (check:report-actual-result actual-result)
              (check:report-failed expected-result))))))
```

5.4 结尾

goldfish/srfi/srfi-78.scm

△ 11

```
) ; end of begin
) ; end of define-library
```


[goldfish/liii/check.scm](#)

[△ 8](#)

```
) ; end of begin
```

```
) ; end of define-library
```

第 6 章

(liii case)

case*克服了R7RS中case无法处理字符串等的缺点。

6.1 测试

tests/goldfish/liii/case-test.scm1 ▾

```
(import (liii check)
        (liii case))

(check-set-mode! 'report-failed)
```

case*索引

tests/goldfish/liii/case-test.scm

△ 2 ▽

```

; 0 clause BSD, from S7 repo s7test.scm
(define (scase x)
  (case* x
    ((a b) 'a-or-b)
    ((1 2/3 3.0) => (lambda (a) (* a 2)))
    ((pi) 1 123)
    (("string1" "string2"))
    ((#<symbol?>) 'symbol!)
    (((+ x #<symbol?>)) 'got-list)
    ((#(1 x 3)) 'got-vector)
    (((+ #<>)) 'empty)
    (((* #<x:symbol?> #<x>)) 'got-label)
    (((#<> #<x:> #<x>)) 'repeated)
    (((#<symbol?> #<symbol?>)) 'two)
    (((#<x:> #<x>)) 'pair)
    (((#<x:> #<x>)) 'vector)
    (((#<symbol?> #<...> #<number?>)) 'vectsn)
    (((#<...> #<number?>)) 'vectstart)
    (((#<string?> #<char-whitespace?> #<...>)) 'vectstr)
    (else 'oops)))

(test (scase 3.0) 6.0)
(test (scase 'pi) 123)
(test (scase "string1") "string1")
(test (scase "string3") 'oops)
(test (scase 'a) 'a-or-b)
(test (scase 'abc) 'symbol!)
(test (scase #()) 'oops)
(test (scase '(+ x z)) 'got-list)
(test (scase #(1 x 3)) 'got-vector)
(test (scase '(+ x 3)) 'oops)
(test (scase '(+ x)) 'empty)
(test (scase '(* z z)) 'got-label)
(test (scase '(* z x)) 'oops)
(test (scase '(+ (abs x) (abs x))) 'repeated)
(test (scase '(+ (abs x) (abs y))) 'oops)
(test (scase '(a b)) 'two)
(test (scase '(1 1)) 'pair)
(test (scase '(1 1 2)) 'oops)
(test (scase #(1 1)) 'vector)
(test (scase #(a b c 3)) 'vectsn)
(test (scase #(1 b 2)) 'vectstart)
(test (scase #("asdf" #\space +nan.0 #<eof>)) 'vectstr)
(test (scase #(a 3)) 'vectsn)
(test (scase #(1)) 'vectstart)
(test (scase #("asdf" #\space)) 'vectstr)
(test (scase #("asdf")) 'oops)

```

6.2 结尾

tests/goldfish/liii/case-test.scm

△ 3

(check-report)

第 7 章

(liii string)^{chapter:liii_string}

7.1 许可证

goldfish/liii/string.scm1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

goldfish/srfi/srfi-13.scm1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

tests/goldfish/liii/string-test.scm

1 ▾

```

;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;

```

7.2 接口

goldfish/liii/string.scm

△ 2

```

(define-library (liii string)
  (export
    ; S7 built-in
    string? string-ref string-length
    ; from (scheme base)
    string-copy string-for-each string-map
    ; from (srfi srfi-13)
    string-null? string-join
    string-every string-any
    string-take string-take-right string-drop string-drop-right
    string-pad string-pad-right
    string-trim string-trim-right string-trim-both
    string-prefix? string-suffix?
    string-index string-index-right
    string-contains string-count
    string-reverse
    string-tokenize
  )
  (import (srfi srfi-13)
    (scheme base)
    (liii error))
  (begin
    ) ; end of begin
  ) ; end of library

```

goldfish/srfi/srfi-13.scm

△ 2 ▽

```
(define-library (srfi srfi-13)
  (import
    (scheme base)
    (srfi srfi-1))
  (export
    string-null? string-copy string-join
    string-every string-any
    string-take string-take-right string-drop string-drop-right
    string-pad string-pad-right
    string-trim string-trim-right string-trim-both
    string-prefix? string-suffix?
    string-index string-index-right
    string-contains string-count
    string-reverse
    string-tokenize)
  (begin
```

7.3 测试

tests/goldfish/liii/string-test.scm

△ 2 ▽

```
(import (liii check)
        (liii string))

(check-set-mode! 'report-failed)
```

7.4 内部公共子函数

goldfish/srfi/srfi-13.scm

△ 3 ▽

```
(define (%string-from-range str start_end)
  (cond ((null-list? start_end) str)
        ((= (length start_end) 1)
         (substring str (car start_end)))
        ((= (length start_end) 2)
         (substring str (first start_end) (second start_end)))
        (else (error 'wrong-number-of-args "%string-from-range"))))
```

[测试一下边界条件，这样后面漏测也没关系了。]

goldfish/srfi/srfi-13.scm

△ 4 ▽

```
(define (%make-criterion char/pred?)
  (cond ((char? char/pred?) (lambda (x) (char=? x char/pred?)))
        ((procedure? char/pred?) char/pred?)
        (else (error 'wrong-type-arg "%make-criterion"))))
```

7.5 谓词

string?

索引

`string?` 是一个S7内置的谓词函数，当且仅当参数对象类型是字符串（例如 `"MathAgape"`）时返回 `#t`，否则都返回 `#f`。当参数为符号（例如 `'MathAgape`）、字符（例如 `#/MathAgape`）、数字（例如 `123`）、列表（例如 `'(1 2 3)`）这些非字符串类型时，都返回 `#f`。`string?` 用于确定对象是否可以被当作字符串处理，在需要执行对字符串特定操作时特别有用，避免类型错误。

tests/goldfish/liii/base-test.scm

△ 45 ▽

```
(check (string? "MathAgape") => #t)
(check (string? "") => #t)

(check (string? 'MathAgape) => #f)
(check (string? #/MathAgape) => #f)
(check (string? 123) => #f)
(check (string? '(1 2 3)) => #f)
```

7.6 列表-字符串转换

string->list

索引

`string->list` 是一个S7内置的函数，用于将字符串转换为字符列表。

tests/goldfish/liii/base-test.scm

△ 46 ▽

```
(check (string->list "MathAgape")
=> '(#\M #\a #\t #\h #\A #\g #\a #\p #\e))

(check (string->list "") => '())
```

list->string

索引

`list->string`是一个S7内置的函数，用于将字符列表转换为字符串。

tests/goldfish/liii/base-test.scm

△ 47 ▽

```
(check
  (list->string '(#\M #\a #\t #\h #\A #\g #\a #\p #\e))
  => "MathAgape")

(check (list->string '()) => "")
```

`string-join` (l [delim [grammar]]) => string

索引

实现

goldfish/srfi/srfi-13.scm

△ 5 ▽

```
(define (string-join l . delim+grammar)
  (define (extract-params params-l)
    (cond ((null-list? params-l)
           (list "" 'infix))
          ((and (= (length params-l) 1)
                 (string? (car params-l)))
           (list (car params-l) 'infix))
          ((and (= (length params-l) 2)
                 (string? (first params-l))
                 (symbol? (second params-l)))
           params-l)
          ((> (length params-l) 2)
           (error 'wrong-number-of-args "optional params in string-join"))
          (else (error 'type-error "optional params in string-join"))))

  (define (string-join-sub l delim)
    (cond ((null-list? l) "")
          ((= (length l) 1) (car l))
          (else
           (string-append
            (car l)
            delim
            (string-join-sub (cdr l) delim)))))

  (let* ((params (extract-params delim+grammar))
         (delim (first params))
         (grammar (second params))
         (ret (string-join-sub l delim)))
    (case grammar
      ('infix ret)
      ('strict-infix
       (if (null-list? l)
           (error 'value-error "empty list not allowed")
           ret))
      ('suffix
       (if (null-list? l) "" (string-append ret delim)))
      ('prefix
       (if (null-list? l) "" (string-append delim ret)))
      (else (error 'value-error "invalid grammar")))))
```

测试

tests/goldfish/liii/string-test.scm

△ 3 ▽

```
(check (string-join '("a" "b" "c")) => "abc")

(check (string-join '("a" "b" "c") ":") => "a:b:c")
(check (string-join '("a" "b" "c") ":" 'infix) => "a:b:c")
(check (string-join '("a" "b" "c") ":" 'suffix) => "a:b:c:")
(check (string-join '("a" "b" "c") ":" 'prefix) => ":a:b:c")

(check (string-join '() ":") => "")
(check (string-join '() ":" 'infix) => "")
(check (string-join '() ":" 'prefix) => "")
(check (string-join '() ":" 'suffix) => "")

(check-catch 'value-error (string-join '() ":" 'strict-infix))
(check-catch 'type-error (string-join '() ":" 2))
(check-catch 'value-error (string-join '() ":" 'no-such-grammar))
(check-catch 'wrong-number-of-args (string-join '() ":" 1 2 3))
```

7.7 谓词

string-null?  (str) => boolean

string-null?是一个谓词函数，当且仅当参数字符串为空（即长度为0）时返回#t，否则都返回#f，也就是说，当参数是非字符串或长度不为0的字符串时，返回#f。

goldfish/srfi/srfi-13.scm

△ 6 ▽

```
(define (string-null? str)
  (and (string? str)
        (lambda (x) (= x 0)) (string-length str))))
```

tests/goldfish/liii/string-test.scm

△ 4 ▽

```
(check-true (string-null? ""))

(check-false (string-null? "MathAgape"))

(check-false (string-null? 'not-a-string))
```

string-every 

string-every是一个谓词函数，它接收一个字符串str和一个评估规则criterion为参数，以及最多2个非负整数为可选参数。start_end用于指定搜索的起始位置和结束位置，检查指定范围内字符串中的每个字符是否都满足评估规则criterion。

goldfish/srfi/srfi-13.scm

△ 7 ▽

```
(define (string-every char/pred? str . start+end)
  (define (string-every-sub pred? str)
    (let
      loop ((i 0) (len (string-length str)))
      (or (= i len)
          (and (pred? (string-ref str i))
               (loop (+ i 1) len)))))

  (let ((str-sub (%string-from-range str start+end))
        (criterion (%make-criterion char/pred?)))
    (string-every-sub criterion str-sub)))
```

tests/goldfish/liii/string-test.scm

△ 5 ▽

```
(check-true (string-every #\x "xxxxxx"))
(check-false (string-every #\x "xxx0xx"))

(check-true (string-every char-numeric? "012345"))
(check-false (string-every char-numeric? "012d45"))
```

注意，评估规则 **criterion** 必须是谓词或字符，否则报错。

tests/goldfish/liii/string-test.scm

△ 6 ▽

```
(check-catch 'wrong-type-arg (string-every 1 "012345"))

(check
  (catch 'wrong-type-arg
    (lambda ()
      (string-every
        #\012345
        "012345")))
    (lambda args #t))
=>
#t)

(check
  (catch 'wrong-type-arg
    (lambda ()
      (string-every
        "012345"
        "012345")))
    (lambda args #t))
=>
#t)
```

注意，谓词要使用字符属性测试函数（例如 **char-numeric?**），不要用类型检查函数（例如 **number?**），否则失去了检查的意义。

tests/goldfish/liii/string-test.scm

△ 7 ▽

```
(check-true (string-every char-numeric? "012345"))
(check-false (string-every number? "012345"))
```

对于可选参数 **start_end**：没有可选参数时，默认搜索整个字符串；只有1个可选参数时，该数指定的是起始位置，结束位置默认为字符串的最后一个字符。注意，有2个可选参数时，第二个数不能小于第一个数，否则报错；指定的起始位置和结束位置不能超过字符串 **str** 的起始范围，否则报错；可选参数超过2个时，报错。（注意，可选参数的规则适用于后面所有调用到 **%string-from-range** 的函数）

tests/goldfish/liii/string-test.scm

△ 8 ▽

```
(check-true (string-every char-numeric? "ab2345" 2))
(check-false (string-every char-numeric? "ab2345" 1))
(check-false (string-every char-numeric? "ab234f" 2))
```

```
(check-true (string-every char-numeric? "ab234f" 2 4))
(check-true (string-every char-numeric? "ab234f" 2 2))
```

```
(check
  (string-every
    char-numeric?
    "ab234f"
    1
    4)
  =>
  #f)
```

```
(check
  (string-every
    char-numeric?
    "ab234f"
    2
    5)
  =>
  #t)
```

```
(check
  (string-every
    char-numeric?
    "ab234f"
    2
    6)
  =>
  #f)
```

```
(check
  (catch 'out-of-range
    (lambda ()
      (string-every
        char-numeric?
        "ab234f"
        2
        7))
    (lambda args #t)))
  =>
  #t)
```

```
(check
  (catch 'out-of-range
    (lambda ()
      (string-every
        char-numeric?
        "ab234f"
        2
        1))
    (lambda args #t)))
  =>
  #t)
```

索引

string-any

`string-any`是一个谓词函数，它接收一个谓词和一个字符串为参数，用于检查字符串中是否存在字符都满足谓词。注意，这里的谓词必须是字符属性测试函数，否则报错。

goldfish/srfi/srfi-13.scm

△ 8 ▽

```
(define (string-any char/pred? str . start+end)
  (define (string-any-sub pred? str)
    (let
      (loop ((i 0) (len (string-length str)))
        (if (= i len)
            #f
            (or (pred? (string-ref str i))
                (loop (+ i 1) len))))))

  (let ((str_sub (%string-from-range str start+end))
        (criterion (%make-criterion char/pred?)))
    (string-any-sub criterion str_sub)))
```

tests/goldfish/liii/string-test.scm

△ 9 ▽

```
(check-true (string-any #\0 "xxx0xx"))
(check-false (string-any #\0 "xxxxxx"))
(check-true (string-any char-numeric? "xxx0xx"))
(check-false (string-any char-numeric? "xxxxxx"))
```

注意，评估规则 `criterion` 必须是谓词或字符，否则报错。

tests/goldfish/liii/string-test.scm

△ 10 ▽

```
(check
  (catch 'wrong-type-arg
    (lambda ()
      (string-every
        0
        "xxx0xx"))
    (lambda args #t)))
=>
#t)

(check
  (catch 'wrong-type-arg
    (lambda ()
      (string-any
        (lambda (n) (= n 0))
        "xxx0xx"))
    (lambda args #t)))
=>
#t)

(check
  (catch 'wrong-type-arg
    (lambda ()
      (string-every
        "0"
        "xxx0xx"))
    (lambda args #t)))
=>
#t)
```

对于可选参数:

tests/goldfish/liii/string-test.scm

△ 11 ▽

```
(check-true (string-any char-alphabetic? "01c345" 2))

(check-false (string-any char-alphabetic? "01c345" 3))

(check
  (string-any
    char-alphabetic?
    "01c345"
    2
    4)
  =>
  #t)

(check
  (string-any
    char-alphabetic?
    "01c345"
    2
    2)
  =>
  #f)

(check
  (string-any
    char-alphabetic?
    "01c345"
    3
    4)
  =>
  #f)

(check
  (string-any
    char-alphabetic?
    "01c345"
    2
    6)
  =>
  #t)

(check
  (catch 'out-of-range
    (lambda ()
      (string-any
        char-alphabetic?
        "01c345"
        2
        7)))
    (lambda args #t))
  =>
  #t)

(check
  (catch 'out-of-range
    (lambda ()
      (string-any
        char-alphabetic?
```

7.8 选择器

S7 内置 索引

string-length

`string-length`是一个S7内置的函数，它接收一个字符串作为参数，返回一个表示该字符串长度的整数，即字符串中包含的字符数。当参数不是字符串，报错。

tests/goldfish/liii/base-test.scm

△ 48 ▾

```
(check (string-length "MathAgape") => 9)
(check (string-length "") => 0)

(check
  (catch 'wrong-type-arg
    (lambda () (string-length 'not-a-string))
    (lambda args #t))
  =>
  #t)
```

S7 内置 索引

string-ref

`string-ref`是一个S7内置的函数，接收一个字符串和一个称为索引值的非负整数k为参数，通过索引k返回字符串的第k个元素（从0开始计数）。当参数为空字符串时，报错。当k为负数，报错。当k大于等于字符串中字符数时，报错。

tests/goldfish/liii/base-test.scm

△ 49 ▾

```
(check (string-ref "MathAgape" 0) => #\M)
(check (string-ref "MathAgape" 2) => #\t)

(check
  (catch 'out-of-range
    (lambda () (string-ref "MathAgape" -1))
    (lambda args #t))
  =>
  #t)

(check
  (catch 'out-of-range
    (lambda () (string-ref "MathAgape" 9))
    (lambda args #t))
  =>
  #t)

(check
  (catch 'out-of-range
    (lambda () (string-ref "" 0))
    (lambda args #t))
  =>
  #t)
```

S7 内置 索引

string-copy

`string-copy`是一个S7内置的函数，用于创建一个现有字符串的副本。这个函数返回一个与原始字符串内容完全相同的新字符串，但它们在内存中是两个独立的实体。也就是说，原始字符串和副本字符串内容相同，但它们是不同的对象，这可以用`equal?`和`eq?`对比出来。

`string-copy`和`%string-from-range`的实现比较接近，区别在于：在没有指定字符串的范围的时候，我们需要使用`substring`来模拟整个字符串的拷贝。S7内置的函数只提供了整个字符串的拷贝，R7RS和SRFI-13是需要额外的字符串的范围的，所以需要重新实现。

goldfish/scheme/base.scm

△ 26 ▽

```
(define (string-copy str . start_end)
  (cond ((null? start_end)
        (substring str 0))
        ((= (length start_end) 1)
         (substring str (car start_end)))
        ((= (length start_end) 2)
         (substring str (car start_end) (cadr start_end)))
        (else (error 'wrong-number-of-args))))
```

测试

tests/goldfish/liii/string-test.scm

△ 12 ▽

```
(define original-string "MathAgape")
(define copied-string (string-copy original-string))

(check-true (equal? original-string copied-string))
(check-false (eq? original-string copied-string))

(check-true
  (equal? (string-copy "MathAgape" 4)
          (string-copy "MathAgape" 4)))

(check-false
  (eq? (string-copy "MathAgape" 4)
       (string-copy "MathAgape" 4)))

(check-true
  (equal? (string-copy "MathAgape" 4 9)
          (string-copy "MathAgape" 4 9)))

(check-false
  (eq? (string-copy "MathAgape" 4 9)
       (string-copy "MathAgape" 4 9)))
```


string-take

索引

string-take是一个函数，接收一个字符串和一个非负整数k为参数，返回字符串的前k个字符组成的新字符串。当字符串字符数量不足k个，报错。

goldfish/srfi/srfi-13.scm

△ 9 ▾

```
(define (string-take str k)
  (list->string (take (string->list str) k)))
```

tests/goldfish/liii/string-test.scm

△ 13 ▾

```
(check (string-take "MathAgape" 4) => "Math")

(check
  (catch 'wrong-type-arg
    (lambda () (string-take "MathAgape" 20))
    (lambda args #t))
  =>
  #t)
```

string-take-right

索引

string-take-right是一个函数，接收一个字符串和一个非负整数k为参数，取出字符串的后k个字符组成新字符串，返回这个新字符串。当字符串字符数量不足k个，报错。

goldfish/srfi/srfi-13.scm

△ 10 ▾

```
(define (string-take-right str k)
  (list->string (take-right (string->list str) k)))
```

tests/goldfish/liii/string-test.scm

△ 14 ▾

```
(check (string-take-right "MathAgape" 1) => "e")

(check-catch 'wrong-type-arg (string-take-right "MathAgape" 20))
```

string-drop

索引

string-drop是一个函数，接收一个字符串和一个非负整数k为参数，返回去掉字符串前k个字符组成的新字符串。当字符串字符数量不足k个，报错。

goldfish/srfi/srfi-13.scm

△ 11 ▾

```
(define (string-drop str k)
  (list->string (drop (string->list str) k)))
```

tests/goldfish/liii/string-test.scm

△ 15 ▾

```
(check (string-drop "MathAgape" 8) => "e")

(check-catch 'wrong-type-arg (string-drop "MathAgape" 20))
```

string-drop-right

索引

string-drop-right是一个函数，接收一个字符串和一个非负整数k为参数，去掉字符串的后k个字符组成新字符串，返回这个新字符串。当字符串字符数量不足k个，报错。当k为负数，报错。

goldfish/srfi/srfi-13.scm

△ 12 ▽

```
(define (string-drop-right str k)
  (list->string (drop-right (string->list str) k)))
```

tests/goldfish/liii/string-test.scm

△ 16 ▽

```
(check (string-drop-right "MathAgape" 5) => "Math")

(check-catch 'wrong-type-arg (string-drop "MathAgape" 20))
```

string-pad

索引

`string-pad`是一个函数，接收一个字符串、一个指定长度（非负整数）为参数，从左填充空格字符直到指定长度，返回这个新字符。当指定长度等于字符串长度，返回的字符串和原字符串内容相同。当指定长度小于字符串长度，则从左去掉字符直到指定长度，返回这个新字符。当参数的指定长度为负数时，报错。

goldfish/srfi/srfi-13.scm

△ 13 ▽

```
(define (string-pad str len . char+start+end)
  (define (string-pad-sub str len ch)
    (let ((orig-len (string-length str)))
      (if (< len orig-len)
          (string-take-right str len)
          (string-append (make-string (- len orig-len) ch) str))))

  (cond ((null-list? char+start+end)
         (string-pad-sub str len #\ ))
        ((list? char+start+end)
         (string-pad-sub
          (%string-from-range str (cdr char+start+end))
          len
          (car char+start+end)))
        (else (error 'wrong-type-arg "string-pad"))))
```

tests/goldfish/liii/string-test.scm

△ 17 ▽

```
(check
  (string-pad "MathAgape" 15)
  =>
  "          MathAgape")

(check
  (string-pad "MathAgape" 12 #\1)
  =>
  "111MathAgape")

(check
  (string-pad "MathAgape" 6 #\1 0 4)
  =>
  "11Math")

(check
  (string-pad "MathAgape" 9)
  =>
  "MathAgape")

(check
  (string-pad "MathAgape" 5)
  =>
  "Agape")

(check
  (string-pad "MathAgape" 2 #\1 0 4)
  =>
  "th")

(check
  (catch 'wrong-type-arg
    (lambda () (string-pad "MathAgape" -1))
    (lambda args #t))
  =>
  #t)
```

string-pad-right

[索引](#)

`string-pad-right` 是一个函数，接收一个字符串、一个指定长度（非负整数）为参数，从右填充空格字符直到指定长度，返回这个新字符。当指定长度等于字符串长度，返回的字符串和原字符串内容相同。当指定长度小于字符串长度，则从右去掉字符直到指定长度，返回这个新字符。当参数的指定长度为负数时，报错。

goldfish/srfi/srfi-13.scm

△ 14 ▽

```

(define (string-pad-right str len . char+start+end)
  (define (string-pad-right-sub str len ch)
    (let ((orig-len (string-length str)))
      (if (< len orig-len)
          (string-take str len)
          (string-append str (make-string (- len orig-len) ch)))))

  (cond ((null-list? char+start+end)
         (string-pad-right-sub str len #\ ))
        ((list? char+start+end)
         (string-pad-right-sub
          (%string-from-range str (cdr char+start+end))
          len
          (car char+start+end)))
        (else (error 'wrong-type-arg "string-pad")))))

```

测试

tests/goldfish/liii/string-test.scm

△ 18 ▽

```

(check (string-pad-right "MathAgape" 15) => "MathAgape      ")

(check (string-pad-right "MathAgape" 12 #\1) => "MathAgape111")

(check
  (string-pad-right "MathAgape" 6 #\1 0 4)
  =>
  "Math11")

(check
  (string-pad-right "MathAgape" 9)
  =>
  "MathAgape")

(check
  (string-pad-right "MathAgape" 9 #\1)
  =>
  "MathAgape")

(check
  (string-pad-right "MathAgape" 4)
  =>
  "Math")

(check
  (string-pad "MathAgape" 2 #\1 0 4)
  =>
  "th")

(check
  (catch 'wrong-type-arg
    (lambda () (string-pad-right "MathAgape" -1))
    (lambda args #t))
  =>
  #t)

```

string-trim

`string-trim`是一个函数，用于去除字符串左端的空白字符。空白字符通常包括空格、制表符、换行符等。当原字符串只包含空白字符，返回空字符串。

[goldfish/srfi/srfi-13.scm](#)
[△ 15 ▽](#)

```
(define (%trim-do str string-trim-sub criterion+start+end)
  (cond ((null-list? criterion+start+end)
        (string-trim-sub str #\ ))
        ((list? criterion+start+end)
         (if (char? (car criterion+start+end))
             (string-trim-sub
              (%string-from-range str (cdr criterion+start+end))
              (car criterion+start+end))
             (string-trim-sub
              (%string-from-range str criterion+start+end)
              #\ )))
        (else (error 'wrong-type-arg "string-trim"))))

(define (string-trim str . criterion+start+end)
  (define (string-trim-sub str space-or-char)
    (let loop ((i 0)
              (len (string-length str)))
      (if (or (= i len) (not (char=? space-or-char (string-ref str i))))
          (substring str i len)
          (loop (+ i 1) len))))
  (%trim-do str string-trim-sub criterion+start+end))
```

tests/goldfish/liii/string-test.scm

△ 19 ▽

```

(check (string-trim " 2 4 ") => "2 4 ")
(check (string-trim " 2 4 " 2) => "2 4 ")
(check (string-trim " 2 4 " 3) => "4 ")
(check (string-trim " 2 4 " 4) => "4 ")
(check (string-trim " 2 4 " 5) => "")

(check-catch 'out-of-range (string-trim " 2 4 " 8))

(check
  (string-trim " 2 4 " 0 4)
  =>
  "2 ")

(check
  (string-trim " 2 4 " 0 7)
  =>
  "2 4 ")

(check
  (catch 'out-of-range
    (lambda () (string-trim " 2 4 " 0 8))
    (lambda args #t))
  =>
  #t)

(check
  (string-trim " 2 4 " #\ )
  =>
  "2 4 ")

(check
  (string-trim "-- 2 4 --" #\-)
  =>
  " 2 4 --")

(check
  (string-trim " - 345" #\ - 1)
  =>
  " 345")

(check
  (string-trim " - 345" #\ - 1 4)
  =>
  " 3")

```

string-trim-right

索引

`string-trim-right` 是一个函数，用于去除字符串右端的空白字符。当原字符串只包含空白字符，返回空字符串。

[goldfish/srfi/srfi-13.scm](#)

△ 16 ▽

```
(define (string-trim-right str . criterion+start+end)
  (define (string-trim-right-sub str space-or-char)
    (let loop ((i (- (string-length str) 1)))
      (cond ((negative? i) "")
            ((char=? space-or-char (string-ref str i)) (loop (- i 1)))
            (else (substring str 0 (+ i 1))))))
  (%trim-do str string-trim-right-sub criterion+start+end))
```

tests/goldfish/liii/string-test.scm

△ 20 ▽

```
(check (string-trim-right " 2 4 ") => " 2 4")
```

```
(check
  (string-trim-right " 2 4 " 1)
  =>
  " 2 4")
```

```
(check
  (string-trim-right " 2 4 " 2)
  =>
  "2 4")
```

```
(check
  (string-trim-right " 2 4 " 3)
  =>
  " 4")
```

```
(check
  (string-trim-right " 2 4 " 4)
  =>
  "4")
```

```
(check
  (string-trim-right " 2 4 " 5)
  =>
  "")
```

```
(check
  (string-trim-right " 2 4 " 6)
  =>
  "")
```

```
(check
  (string-trim-right " 2 4 " 7)
  =>
  "")
```

```
(check
  (catch 'out-of-range
    (lambda () (string-trim-right " 2 4 " 8))
    (lambda args #t))
  =>
  #t)
```

```
(check (string-trim-right " 2 4 " 0 4) => " 2")
(check (string-trim-right " 2 4 " 0 7) => " 2 4")
```

```
(check
  (catch 'out-of-range
    (lambda () (string-trim-right " 2 4 " 0 8))
    (lambda args #t))
  =>
  #t)
```

```
(check (string-trim-right " 2 4 " #\ ) => " 2 4")
(check (string-trim-right "-- 2 4 --" #\-) => "-- 2 4 ")
(check (string-trim-right "012-" #\ - 1) => "12")
(check (string-trim-right "012-4" #\ - 0 4) => "012")
```


string-trim-both



`string-trim-both`是一个函数，用于去除字符串两端的空白字符。当原字符串只包含空白字符，报错。

goldfish/srfi/srfi-13.scm

△ 17 ▾

```
(define (string-trim-both str . criterion+start+end)
  (define (string-trim-both-sub str space-or-char)
    (let loop ((i 0)
               (len (string-length str)))
      (if (or (= i len) (not (char=? space-or-char (string-ref str i))))
          (let loop-end ((j (- len 1)))
            (if (or (< j 0) (not (char=? space-or-char (string-ref str j))))
                (substring str i (+ j 1))
                (loop-end (- j 1))))
          (loop (+ i 1) len))))
  (%trim-do str string-trim-both-sub criterion+start+end))
```

tests/goldfish/liii/string-test.scm

△ 21 ▾

```
(check (string-trim-both " 2 4 ") => "2 4")
(check (string-trim-both "--2 4--" #\ ) => "2 4")
```

7.9 前缀和后缀

string-prefix?



`string-prefix?`是一个谓词函数，用于检查一个字符串是否是另一个字符串的前缀。若第一个字符串是第二个字符串的前缀，则函数返回`#t`；否则返回`#f`。特别地，空字符串是任意字符串的前缀；两个内容相同的字符串互为前缀。

goldfish/srfi/srfi-13.scm

△ 18 ▾

```
(define (string-prefix? prefix str)
  (let* ((prefix-len (string-length prefix))
        (str-len (string-length str)))
    (and (<= prefix-len str-len)
         (let loop ((i 0))
           (or (= i prefix-len)
               (and (char=? (string-ref prefix i)
                             (string-ref str i))
                    (loop (+ i 1))))))))
```

tests/goldfish/liii/string-test.scm

△ 22 ▾

```
(check-true (string-prefix? "Ma" "MathAgape"))
(check-true (string-prefix? "" "MathAgape"))
(check-true (string-prefix? "MathAgape" "MathAgape"))
(check-false (string-prefix? "a" "MathAgape"))
```

string-suffix?



`string-suffix?`是一个谓词函数，用于检查一个字符串是否是另一个字符串的后缀。若第一个字

符串是第二个字符串的后缀，则函数返回`#t`；否则返回`#f`。特别地，空字符串是任意字符串的后缀；两个内容相同的字符串互为后缀。

goldfish/srfi/srfi-13.scm

△ 19 ▾

```
(define (string-suffix? suffix str)
  (let* ((suffix-len (string-length suffix))
        (str-len (string-length str)))
    (and (<= suffix-len str-len)
         (let loop ((i 0)
                    (j (- str-len suffix-len)))
           (or (= i suffix-len)
               (and (char=? (string-ref suffix i)
                             (string-ref str j))
                    (loop (+ i 1) (+ j 1))))))))
```

tests/goldfish/liii/string-test.scm

△ 23 ▾

```
(check-true (string-suffix? "e" "MathAgape"))
(check-true (string-suffix? "" "MathAgape"))
(check-true (string-suffix? "MathAgape" "MathAgape"))
(check-false (string-suffix? "p" "MathAgape"))
```

7.10 搜索

string-index

索引

goldfish/srfi/srfi-13.scm

△ 20 ▾

```
(define (string-index str char/pred? . start+end)
  (define (string-index-sub str pred?)
    (let loop ((i 0))
      (cond ((>= i (string-length str)) #f)
            ((pred? (string-ref str i)) i)
            (else (loop (+ i 1))))))

  (let* ((start (if (null-list? start+end) 0 (car start+end)))
        (str-sub (%string-from-range str start+end))
        (pred? (%make-criterion char/pred?)))
    (ret (string-index-sub str-sub pred?)))
  (if ret (+ start ret) ret)))
```

tests/goldfish/liii/string-test.scm

△ 24 ▾

```
(check (string-index "0123456789" #\2) => 2)
(check (string-index "0123456789" #\2 2) => 2)
(check (string-index "0123456789" #\2 3) => #f)
(check (string-index "01x3456789" char-alphabetic?) => 2)
```

string-index-right

索引

goldfish/srfi/srfi-13.scm

△ 21 ▾

```
(define (string-index-right str char/pred? . start+end)
  (define (string-index-right-sub str pred?)
    (let loop ((i (- (string-length str) 1)))
      (cond ((< i 0) #f)
            ((pred? (string-ref str i)) i)
            (else (loop (- i 1))))))

  (let* ((start (if (null-list? start+end) 0 (car start+end)))
        (str-sub (%string-from-range str start+end))
        (pred? (%make-criterion char/pred?)))
    (ret (string-index-right-sub str-sub pred?)))
  (if ret (+ start ret) ret)))
```

tests/goldfish/liii/string-test.scm

△ 25 ▾

```
(check (string-index-right "0123456789" #\8) => 8)
(check (string-index-right "0123456789" #\8 2) => 8)
(check (string-index-right "0123456789" #\8 9) => #f)
(check (string-index-right "01234567x9" char-alphabetic?) => 8)
```

string-contains

索引

`string-contains` 是一个函数，用于检查一个字符串是否包含另一个子字符串，包含则返回 `#t`，返回 `#f`。

goldfish/srfi/srfi-13.scm

△ 22 ▾

```
(define (string-contains str sub-str)
  (let loop ((i 0))
    (let ((len (string-length str))
          (sub-str-len (string-length sub-str)))
      (if (> i (- len sub-str-len))
          #f
          (if (string=?
                (substring
                 str
                 i
                 (+ i sub-str-len))
                sub-str)
              #t
              (loop (+ i 1)))))))
```

tests/goldfish/liii/string-test.scm

△ 26 ▾

```
(check-true (string-contains "0123456789" "3"))
(check-true (string-contains "0123456789" "34"))
(check-false (string-contains "0123456789" "24"))
```

string-count

索引

goldfish/srfi/srfi-13.scm

△ 23 ▾

```
(define (string-count str char/pred? . start+end)
  (let ((str-sub (%string-from-range str start+end))
        (criterion (%make-criterion char/pred?)))
    (count criterion (string->list str-sub))))
```

tests/goldfish/liii/string-test.scm

△ 27 ▾

```
(check (string-count "xyz" #\x) => 1)
(check (string-count "xyz" #\x 0 1) => 1)
(check (string-count "xyz" #\y 0 1) => 0)
(check (string-count "xyz" #\x 0 3) => 1)
(check (string-count "xyz" (lambda (x) (char=? x #\x))) => 1)
```

7.11 翻转和追加

string-reverse

索引

goldfish/srfi/srfi-13.scm

△ 24 ▾

```
(define (string-reverse str . start+end)
  (cond ((null-list? start+end)
        (reverse str))
        ((= (length start+end) 1)
         (let ((start (first start+end)))
           (string-append (substring str 0 start)
                          (reverse (substring str start))))))
        ((= (length start+end) 2)
         (let ((start (first start+end))
               (end (second start+end)))
           (string-append (substring str 0 start)
                          (reverse (substring str start end))
                          (substring str end))))
        (else (error 'wrong-number-of-args "string-reverse"))))
```

tests/goldfish/liii/string-test.scm

△ 28 ▾

```
(check (string-reverse "01234") => "43210")

(check-catch 'out-of-range (string-reverse "01234" -1))

(check (string-reverse "01234" 0) => "43210")
(check (string-reverse "01234" 1) => "04321")
(check (string-reverse "01234" 5) => "01234")

(check-catch 'out-of-range (string-reverse "01234" 6))

(check (string-reverse "01234" 0 2) => "10234")
(check (string-reverse "01234" 1 3) => "02134")
(check (string-reverse "01234" 1 5) => "04321")
(check (string-reverse "01234" 0 5) => "43210")

(check-catch 'out-of-range (string-reverse "01234" 1 6))

(check-catch 'out-of-range (string-reverse "01234" -1 3))
```

string-append

索引

string-append

`string-append`是一个S7内置的函数，用于连接两个或多个字符串。它会将所有提供的字符串参数逐个拼接在一起，并返回一个新的字符串，原始字符串不会被修改。当没有参数时，返回空字符串。

tests/goldfish/liii/base-test.scm

△ 50 ▾

```
(check (string-append "Math" "Agape") => "MathAgape")
```

```
(check (string-append) => "")
```

7.12 Fold, unfold & map

7.12.1

string-map

索引

实现

goldfish/scheme/base.scm

△ 27 ▾

```
(define (string-map p . args) (apply string (apply map p args)))
```

测试

tests/goldfish/liii/string-test.scm

△ 29 ▽

```
(check
  (string-map
    (lambda (ch) (integer->char (+ 1 (char->integer ch)))))
    "HAL")
=> "IBM")
```

718 **string-for-each** (proc str1 [str2 ...]) 索引

goldfish/scheme/base.scm

△ 28 ▽

```
(define string-for-each for-each)
```

tests/goldfish/liii/string-test.scm

△ 30 ▽

```
(check
  (let ((lst '()))
    (string-for-each
      (lambda (x) (set! lst (cons (char->integer x) lst)))
      "12345")
    lst)
=> '(53 52 51 50 49))

(check
  (let ((lst '()))
    (string-for-each
      (lambda (x) (set! lst (cons (- (char->integer x) (char->integer #\0)) lst)))
      "12345")
    lst)
=> '(5 4 3 2 1))

(check
  (let ((lst '()))
    (string-for-each
      (lambda (x) (set! lst (cons (- (char->integer x) (char->integer #\0)) lst)))
      "123")
    lst)
=> '(3 2 1))

(check
  (let ((lst '()))
    (string-for-each
      (lambda (x) (set! lst (cons (- (char->integer x) (char->integer #\0)) lst)))
      "")
    lst)
=> '())
```

7.13 插入和解析

string-tokenize 索引

实现

goldfish/srfi/srfi-13.scm

△ 25 ▾

```

(define (string-tokenize str . char+start+end)

  (define (string-tokenize-sub str char)

    (define (tokenize-helper tokens cursor)
      (let ((sep-pos/false (string-index str char cursor)))
        (if (not sep-pos/false)
            (reverse (cons (substring str cursor) tokens))
            (let ((new-tokens
                    (if (= cursor sep-pos/false)
                        tokens
                        (cons (substring str cursor sep-pos/false) tokens))))
              (next-cursor (+ sep-pos/false 1)))
              (tokenize-helper new-tokens next-cursor))))))

    (tokenize-helper '() 0))

  (cond ((null-list? char+start+end)
        (string-tokenize-sub str #\ ))
        ((list? char+start+end)
         (string-tokenize-sub
          (%string-from-range str (cdr char+start+end)
                              (car char+start+end)))
         (else (error 'wrong-type-arg "string-tokenize"))))

```

测试

tests/goldfish/liii/string-test.scm

△ 31 ▾

```

(check
  (string-tokenize "1 22 333")
  => '("1" "22" "333"))

(check
  (string-tokenize "1 22 333" #\2)
  => '("1 " " 333"))

(check
  (string-tokenize "1 22 333" #\ 2)
  => '("22" "333"))

```

7.14 结尾

goldfish/srfi/srfi-13.scm

△ 26

```

) ; end of begin
) ; end of define-library

```

tests/goldfish/liii/string-test.scm

△ 32

```

(check-report)

```

第 8 章

(liii list)

chapter:liii_list

8.1 许可证

goldfish/liii/list.scm 1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

tests/goldfish/liii/list-test.scm 1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

goldfish/srfi/srfi-1.scm

1 ▾

```

;;; SRFI-1 list-processing library                                -*- Scheme -*-
;;; Reference implementation
;;;
;;; SPDX-License-Identifier: MIT
;;;
;;; Copyright (c) 1998, 1999 by Olin Shivers. You may do as you please with
;;; this code as long as you do not remove this copyright notice or
;;; hold me liable for its use. Please send bug reports to shivers@ai.mit.edu.
;;;      -Olin
;;;
;;; Copyright (c) 2024 The Goldfish Scheme Authors
;;; Follow the same License as the original one

```

8.2 接口

Liii List函数库是金鱼标准库中的List函数库，其中的大部分函数来自函数库(`srfi srfi-1`)，小部分是三鲤自己设计的函数。来自SRFI 1的，我们只是在Liii List中导出相关函数名，相关实现和单元测试都在SRFI 1中维护。

goldfish/liii/list.scm

△ 2 ▾

```

(define-library (liii list)
  (export
    ; S7 built-in
    cons car cdr map for-each
    ; SRFI 1: Constructors
    circular-list iota
    ; SRFI 1: Predicates
    null-list? circular-list?
    ; SRFI 1: Selectors
    first second third fourth fifth sixth seventh eighth ninth tenth
    take drop take-right drop-right
    last-pair last
    ; SRFI 1: fold, unfold & map
    count fold fold-right reduce reduce-right
    filter partition remove append-map
    ; SRFI 1: Searching
    find any every list-index
    take-while drop-while
    ; SRFI 1: deleting
    delete
    ; Liii List extensions
    list-view flatmap
    list-null? list-not-null? not-null-list?
    length=? length>? length>=?
  )
  (import (srfi srfi-1)
          (liii error))
  (begin

```

goldfish/srfi/srfi-1.scm

△ 2 ▽

```
(define-library (srfi srfi-1)
(import (liii error)
        (liii base))
(export
  circular-list iota circular-list? null-list?
  first second third fourth fifth
  sixth seventh eighth ninth tenth
  take drop take-right drop-right count fold fold-right
  reduce reduce-right append-map filter partition remove find
  delete delete-duplicates
  take-while drop-while list-index any every
  last-pair last)
(begin
```

8.3 测试

在金鱼Scheme中的SRFI 1实现需要遵循最小依赖原则，目前`delete-duplicates`是一个复杂度比较高的实现，在SRFI 1中保留，但并不在`(liii list)`导出，故而在本测试文件的开头需要从`(srfi srfi-1)`单独导入。

tests/goldfish/liii/list-test.scm

△ 2 ▽

```
(import (liii list)
        (liii check)
        (only (srfi srfi-1) delete-duplicates))

(check-set-mode! 'report-failed)
```

8.4 SRFI-1

SRFI-1中有一部分函数已经在R7RS的`(scheme base)`库里面了。本节包含了R7RS定义的`(scheme base)`里面和列表相关的函数，这些函数的实现和测试均在`base.scm`和`base-test.scm`中维护。用户可以通过`(import (liii base))`导入这些函数，或者使用`(import (liii list))`导入这些函数。

8.4.1 构造器

 **cons**

 索引

S7 Scheme内置的R7RS中定义的函数。

list

 索引

S7 Scheme内置的R7RS中定义的函数。

make-list  `(make-list k [fill]) => list`

S7 Scheme内置的R7RS中定义的函数。

tests/goldfish/liii/base-test.scm

△ 51 ▽

```
(check (make-list 3 #\a) => (list #\a #\a #\a))
(check (make-list 3) => (list #f #f #f))

(check (make-list 0) => (list ))
```

circular-list

索引

goldfish/srfi/srfi-1.scm

△ 3 ▽

```
; 0 clause BSD, from S7 repo stuff.scm
(define circular-list
  (lambda (objs)
    (let ((lst (copy objs)))
      (set-cdr! (list-tail lst (- (length lst) 1)) lst))))
```

iota

索引

goldfish/srfi/srfi-1.scm

△ 4 ▽

```
; 0 clause BSD, from S7 repo stuff.scm
(define* (iota n (start 0) (incr 1))
  (when (not (integer? n))
    (type-error "iota: n must be a integer"))
  (when (< n 0)
    (value-error "iota: n must be postive but received ~d" n))
  (let ((lst (make-list n)))
    (do ((p lst (cdr p))
        (i start (+ i incr)))
        ((null? p) lst)
      (set! (car p) i))))
```

测试用例

tests/goldfish/liii/list-test.scm

△ 3 ▽

```
(check (iota 3) => (list 0 1 2))
(check (iota 3 7) => (list 7 8 9))
(check (iota 2 7 2) => (list 7 9))

(check-catch 'value-error (iota -1))
(check-catch 'type-error (iota 'a))
```

8.4.2 谓词

S7 repo

pair? (obj) => bool

索引

pair?是S7 Scheme内置的函数：当且仅当obj是序对时，返回真。

tests/goldfish/liii/base-test.scm

△ 52 ▽

```
(check (pair? '(a . b)) => #t)
(check (pair? '(a b c)) => #t)
(check (pair? '()) => #f)
(check (pair? '#(a b)) => #f)
```

`list?` 索引 (obj) => bool

基本列表测试：检查空列表和包含元素的列表是否被识别为列表。

tests/goldfish/liii/base-test.scm

△ 53 ▽

```
(check-true (list? '()))
(check-true (list? '(a)))
(check-true (list? '(a b c)))
(check-true (list? '(1 . 2)))
(check-true (list? '(1 2 . 3)))
```

嵌套列表测试：检查包含其他列表作为元素的列表是否被正确识别。

tests/goldfish/liii/base-test.scm

△ 54 ▽

```
(check-true (list? '((a) (b) (c))))
(check-true (list? '(a (b) c)))
```

循环列表测试：检查函数是否能够正确处理循环列表结构。

tests/goldfish/liii/base-test.scm

△ 55 ▽

```
(check-true (list? (let ((x '(1 2 3))) (set-cdr! (cddr x) x) x))))
```

非列表测试：检查基本数据类型以及向量是否被正确地识别为非列表。

tests/goldfish/liii/base-test.scm

△ 56 ▽

```
(check-false (list? #t))
(check-false (list? #f))
(check-false (list? 123))
(check-false (list? "Hello"))
(check-false (list? '#(1 2 3)))
(check-false (list? '#()))
(check-false (list? '12345))
```

`null?` 索引 (obj) => bool

`null?`是S7 Scheme内置的函数：当且仅当obj是空列表的时候，返回值为真。

tests/goldfish/liii/base-test.scm

△ 57 ▽

```
(check (null? '()) => #t)
(check (null? '(1)) => #f)
(check (null? '(1 2)) => #f)
```

`circular-list?` 索引

goldfish/srfi/srfi-1.scm

△ 5 ▽

```
; 0 clause BSD, from S7 repo stuff.scm
(define circular-list?
  (lambda (obj)
    (catch #t
      (lambda () (infinite? (length obj)))
      (lambda args #f)))))
```

[在Windows上CI的结果是#f, 需要修复这个问题]

tests/goldfish/liii/list-test.scm

△ 4 ▽

```
; (check (circular-list? (circular-list 1 2)) => #t)
```

null-list?

索引

`null-list?` 是一个函数，返回 `#t` 当且仅当参数是空列表。当参数为空列表，返回 `#t`；否则报错。

goldfish/srfi/srfi-1.scm

△ 6 ▽

```
(define (null-list? l)
  (cond ((pair? l) #f)
        ((null? l) #t)
        (else
         (error 'wrong-type-arg "null-list?: argument out of domain" l))))
```

tests/goldfish/liii/list-test.scm

△ 5 ▽

```
(check (null-list? '()) => #t)
```

当参数为序对，返回 `#f`。

tests/goldfish/liii/list-test.scm

△ 6 ▽

```
(check (null-list? '(1 . 2)) => #f)
```

当参数为非空列表，返回 `#f`。

tests/goldfish/liii/list-test.scm

△ 7 ▽

```
(check (null-list? '(1 2)) => #f)
```

当参数既不是序对也不是列表，报错。

辨析：`null?` 在参数为非序对时，不报错，只是返回 `#f`。

tests/goldfish/liii/list-test.scm

△ 8 ▽

```
(check (null? 1) => #f)
```

如果已经确定需要判别的对象是列表，使用 `null-list?` 更加合适。`null?` 无法分辨非空的序对和非空的列表，命名上偏模糊，不推荐使用。

8.4.3 选择器

索引

car

一个序对由 `<car>` 部分与 `<cdr>` 部分组成，形如：`(<car> . <cdr>)`。`car` 是 S7 Scheme 内置的 R7RS 定义的函数，用于返回序对的 `<car>` 部分。`car` 的参数必须是序对，否则报错；特别地，空列表不是序对，报错。

tests/goldfish/liii/base-test.scm

△ 58 ▽

```
(check (car '(a b c . d)) => 'a)
(check (car '(a b c)) => 'a)

(check-catch 'wrong-type-arg (car '()))
```

cdr

索引


`cdr`是S7 Scheme内置的R7RS定义的函数，用于返回序对的<ampcode;cdr>部分。`cdr`的参数必须是序对，否则报错；特别地，空列表不是序对，报错。

tests/goldfish/liii/base-test.scm

△ 59 ▽

```
(check (cdr '(a b c . d)) => '(b c . d))
(check (cdr '(a b c)) => '(b c))

(check-catch 'wrong-type-arg (cdr '()))
```

caar  `(caar pair)` => obj

`caar`是S7 Scheme内置的R7RS定义的函数，用于返回序对<ampcode;car>部分的<ampcode;car>部分。参数必须是序对，且该序对的<ampcode;car>部分的内容也要是序对，否则报错。特别地，空列表不是序对，报错。

tests/goldfish/liii/base-test.scm

△ 60 ▽

```
(check (caar '((a . b) . c)) => 'a)

(check-catch 'wrong-type-arg (caar '(a b . c)))
(check-catch 'wrong-type-arg (caar '()))
```

list-ref 

`list-ref`是S7 Scheme内置的R7RS定义的函数，接收一个列表和一个称为索引值的非负整数k为参数，通过索引值k返回列表的第k个元素（从0开始计数）。当参数为空列表时，报错。当k为负数，报错。当k大于等于列表中元素数时，报错。

注意`(cons '(1 2) '(3 4))`其实是一个列表，但这个列表的元素不都是整数。这个例子容易理解成结果是一个序对且不是列表。

```
> (cons '(1 2) '(3 4))

((1 2) 3 4)

>
```

tests/goldfish/liii/base-test.scm

△ 61 ▽

```
(check (list-ref (cons '(1 2) '(3 4)) 1) => 3)
```

tests/goldfish/liii/base-test.scm

△ 62 ▽

```
(check (list-ref '(a b c) 2) => 'c)

(check-catch 'wrong-type-arg (list-ref '() 0))

(check-catch 'out-of-range (list-ref '(a b c) -1))
(check-catch 'out-of-range (list-ref '(a b c) 3))
```

first 

`first`是一个函数，`car`的同义词，用于返回列表的第1个元素。当列表元素不足1个，报错。

tests/goldfish/liii/list-test.scm

△ 9 ▽

```
(check (first '(1 2 3 4 5 6 7 8 9 10)) => 1)
(check (first '(left . right)) => 'left)

(check-catch 'wrong-type-arg (first '()))
```

goldfish/srfi/srfi-1.scm

△ 7 ▽

```
(define first car)
```

second

索引

`second`是一个函数，`cadr`的同义词，用于返回列表的第2个元素。当列表元素不足2个，报错。

tests/goldfish/liii/list-test.scm

△ 10 ▾

```
(check (second '(1 2 3 4 5 6 7 8 9 10)) => 2)
```

```
(check-catch 'wrong-type-arg (second '(left . right)))
```

```
(check-catch 'wrong-type-arg (second '(1)))
```

goldfish/srfi/srfi-1.scm

△ 8 ▾

```
(define second cadr)
```

third

索引

`third`是一个函数，`caddr`的同义词，用于返回列表的第3个元素。当列表元素不足3个，报错。

tests/goldfish/liii/list-test.scm

△ 11 ▾

```
(check (third '(1 2 3 4 5 6 7 8 9 10)) => 3)
```

```
(check-catch 'wrong-type-arg (third '(1 2)))
```

goldfish/srfi/srfi-1.scm

△ 9 ▾

```
(define third caddr)
```

fourth

索引

goldfish/srfi/srfi-1.scm

△ 10 ▾

```
(define (fourth x) (list-ref x 3))
```

tests/goldfish/liii/list-test.scm

△ 12 ▾

```
(check (fourth '(1 2 3 4 5 6)) => 4)
```

fifth

索引

goldfish/srfi/srfi-1.scm

△ 11 ▾

```
(define (fifth x) (list-ref x 4))
```

tests/goldfish/liii/list-test.scm

△ 13 ▾

```
(check (fifth '(1 2 3 4 5 6 7 8 9 10)) => 5)
```

sixth

索引

goldfish/srfi/srfi-1.scm

△ 12 ▾

```
(define (sixth x) (list-ref x 5))
```

tests/goldfish/liii/list-test.scm

△ 14 ▾

```
(check (sixth '(1 2 3 4 5 6 7 8 9 10)) => 6)
```

seventh

索引

goldfish/srfi/srfi-1.scm

△ 13 ▾

```
(define (seventh x) (list-ref x 6))
```


tests/goldfish/liii/list-test.scm

△ 15 ▾

```
(check (seventh '(1 2 3 4 5 6 7 8 9 10)) => 7)
```

eighth

索引

goldfish/srfi/srfi-1.scm

△ 14 ▾

```
(define (eighth x) (list-ref x 7))
```

tests/goldfish/liii/list-test.scm

△ 16 ▾

```
(check (eighth '(1 2 3 4 5 6 7 8 9 10)) => 8)
```

ninth

索引

goldfish/srfi/srfi-1.scm

△ 15 ▾

```
(define (ninth x) (list-ref x 8))
```

tests/goldfish/liii/list-test.scm

△ 17 ▾

```
(check (ninth '(1 2 3 4 5 6 7 8 9 10)) => 9)
```

tenth

索引

tenth是一个函数，用于返回列表的第10个元素。当列表元素不足10个，报错。

tests/goldfish/liii/list-test.scm

△ 18 ▾

```
(check (tenth '(1 2 3 4 5 6 7 8 9 10)) => 10)
```

goldfish/srfi/srfi-1.scm

△ 16 ▾

```
(define (tenth x)
  (cadr (cddddr (cddddr x))))
```

take

索引

take是一个函数，接收一个列表和一个非负整数k为参数，返回列表的前k个元素组成的新列表。当列表元素数量不足k个，报错。

测试

tests/goldfish/liii/list-test.scm

△ 19 ▾

```
(check (take '(1 2 3 4) 3) => '(1 2 3))
(check (take '(1 2 3 4) 4) => '(1 2 3 4))
(check (take '(1 2 3 . 4) 3) => '(1 2 3))

(check-catch 'wrong-type-arg (take '(1 2 3 4) 5))
(check-catch 'wrong-type-arg (take '(1 2 3 . 4) 4))
```

实现

goldfish/srfi/srfi-1.scm

△ 17 ▾

```
(define (take l k)
  (let loop ((l l) (k k))
    (if (zero? k) '()
        (cons (car l)
                (loop (cdr l) (- k 1))))))
```

drop

索引

drop是一个函数，接收一个列表和一个非负整数k为参数，返回去掉列表前k个元素组成的新列表。当列表元素数量不足k个，报错。

tests/goldfish/liii/list-test.scm

△ 20 ▽

```
(check (drop '(1 2 3 4) 2) => '(3 4))

(check (drop '(1 2 3 4) 4) => '())

(check-catch 'wrong-type-arg (drop '(1 2 3 4) 5))

(check (drop '(1 2 3 . 4) 3) => 4)

(check-catch 'wrong-type-arg (drop '(1 2 3 . 4) 4))
```

goldfish/srfi/srfi-1.scm

△ 18 ▽

```
(define (drop 1 k)
  (let iter ((l 1) (k k))
    (if (zero? k) 1 (iter (cdr l) (- k 1)))))
```

take-right

索引

take-right是一个函数，接收一个列表和一个非负整数k为参数，取出列表的后k个元素组成新列表，返回这个新列表。当列表元素数量不足k个，报错。

tests/goldfish/liii/list-test.scm

△ 21 ▽

```
(check (take-right '(1 2 3 4) 3) => '(2 3 4))

(check (take-right '(1 2 3 4) 4) => '(1 2 3 4))

(check
  (catch 'wrong-type-arg
    (lambda () (take-right '(1 2 3 4) 5))
    (lambda args #t))
  => #t)

(check (take-right '(1 2 3 . 4) 3) => '(1 2 3 . 4))

(check
  (catch 'wrong-type-arg
    (lambda () (take-right '(1 2 3 . 4) 4))
    (lambda args #t))
  => #t)
```

goldfish/srfi/srfi-1.scm

△ 19 ▽

```
(define (take-right 1 k)
  (let lp ((lag 1) (lead (drop 1 k)))
    (if (pair? lead)
        (lp (cdr lag) (cdr lead))
        lag)))
```

drop-right

索引

`drop-right`是一个函数，接收一个列表和一个非负整数 k 为参数，去掉列表的后 k 个元素组成新列表，返回这个新列表。当列表元素数量不足 k 个，报错。当 k 为负数，报错。

goldfish/srfi/srfi-1.scm

△ 20 ▽

```
(define (drop-right l k)
  (let recur ((lag l) (lead (drop l k)))
    (if (pair? lead)
        (cons (car lag) (recur (cdr lag) (cdr lead)))
        '())))
```

tests/goldfish/liii/list-test.scm

△ 22 ▽

```
(check (drop-right '(1 2 3 4) 2) => '(1 2))

(check (drop-right '(1 2 3 4) 4) => '())

(check
  (catch 'wrong-type-arg
    (lambda () (drop-right '(1 2 3 4) 5))
    (lambda args #t))
  => #t)

(check
  (catch 'wrong-type-arg
    (lambda () (drop-right '(1 2 3 4) -1))
    (lambda args #t))
  => #t)

(check (drop-right '(1 2 3 . 4) 3) => '())

(check
  (catch 'wrong-type-arg
    (lambda () (drop-right '(1 2 3 . 4) 4))
    (lambda args #t))
  => #t)
```

last-pair

`last-pair`是一个函数，以序对形式返回列表的最后一个元素，参数必须是序对，空列表报错。

goldfish/srfi/srfi-1.scm

△ 21 ▽

```
(define (last-pair l)
  (if (pair? (cdr l))
      (last-pair (cdr l)) l))
```

tests/goldfish/liii/list-test.scm

△ 23 ▽

```
(check (last-pair '(a b c)) => '(c))
(check (last-pair '(c)) => '(c))

(check (last-pair '(a b . c)) => '(b . c))
(check (last-pair '(b . c)) => '(b . c))

(check-catch 'wrong-type-arg (last-pair '()))
```

last

`last`是一个函数，以符号形式返回列表的最后一个元素，参数必须是序对，空列表报错。

goldfish/srfi/srfi-1.scm

△ 22 ▾

```
(define (last l)
  (car (last-pair l)))
```

tests/goldfish/liii/list-test.scm

△ 24 ▾

```
(check (last '(a b c)) => 'c)
(check (last '(c)) => 'c)

(check (last '(a b . c)) => 'b)
(check (last '(b . c)) => 'b)

(check-catch 'wrong-type-arg (last '()))
```

8.4.4 常用函数

 `length`

 `length`

`length` (lst) -> integer

`length`是一个S7内置函数，它接收一个列表为参数，返回该列表中元素的数量。如果参数不是列表，返回0。

tests/goldfish/liii/base-test.scm

△ 63 ▾

```
(check (length ()) => 0)
(check (length '(a b c)) => 3)
(check (length '(a (b) (c d e))) => 3)

(check (length 2) => #f)
(check (length '(a . b)) => -1)
```

`append`

 `append`

`append`是一个S7内置的R7RS定义的函数，它接收多个列表为参数，按顺序拼接在一起，返回一个新的列表。`append`没有参数时，返回空列表。

tests/goldfish/liii/base-test.scm

△ 64 ▾

```
(check (append '(a) '(b c d)) => '(a b c d))
(check (append '(a b) 'c) => '(a b . c))

(check (append () 'c) => 'c)
(check (append) => '())
```

`count`

 `count`

`count`是一个高阶函数，它接收两个参数：一个谓词和一个列表；返回满足谓词条件的元素在列表中出现的次数。

goldfish/srfi/srfi-1.scm

△ 23 ▾

```
(define (count pred list1 . lists)
  (let lp ((lis list1) (i 0))
    (if (null-list? lis) i
        (lp (cdr lis) (if (pred (car lis)) (+ i 1) i)))))
```

tests/goldfish/liii/list-test.scm

△ 25 ▾

```
(check (count even? '(3 1 4 1 5 9 2 5 6)) => 3)
```

8.4.5 折叠和映射

 
map

`map`是S7的内置高阶函数，它接收一个函数和一个列表为参数，将该函数应用于该列表的每个元素上，并返回一个新列表。

tests/goldfish/liii/base-test.scm

△ 65 ▾

```
(check (map square (list 1 2 3 4 5)) => '(1 4 9 16 25))
```





`for-each` (`for-each` `proc` `list1` [`list2` ...])

`for-each`是S7内置的高阶函数。`for-each`的参数与`map`的参数类似，但`for-each`调用`proc`是为了它的副作用，而不是为了它的返回值。与`map`不同，`for-each`保证会按照从第一个元素到最后一个元素的顺序调用`proc`，并且`for-each`返回的值是未指定的。如果给出了多个列表，并且不是所有列表的长度都相同，`for-each`会在最短的列表用尽时终止。当`proc`不接受与`lists`数量相同的参数，报错。

tests/goldfish/liii/base-test.scm

△ 66 ▾

```
(check
  (let ((v (make-vector 5)))
    (for-each (lambda (i) (vector-set! v i (* i i)))
              (iota 5))
    v)
=> #(0 1 4 9 16))

(check
  (let ((v (make-vector 5 #f)))
    (for-each (lambda (i) (vector-set! v i (* i i)))
              (iota 4))
    v)
=> #(0 1 4 9 #f))

(check
  (let ((v (make-vector 5 #f)))
    (for-each (lambda (i) (vector-set! v i (* i i)))
              (iota 0))
    v)
=> #( #f #f #f #f #f ))
```

`fold` 

`fold`是一个高阶函数，它接受三个参数：一个函数、一个初始值和一个列表，将函数累积地应用到一个列表的所有元素上，从左到右，从而将列表折叠成一个单一的值。

goldfish/srfi/srfi-1.scm

△ 24 ▾

```
(define (fold f initial l)
  (when (not (procedure? f))
    (error 'type-error "The first param must be a procedure"))
  (if (null? l)
      initial
      (fold f
            (f (car l) initial)
            (cdr l)))))
```

这是SRFI-1官方提供的实现，我们暂时不用。

```
(define (fold kons knil lis1 . lists)
  (if (pair? lists)
      (let lp ((lists (cons lis1 lists)) (ans knil))
        (receive (cars+ans cdrs) (%cars+cdrs+ lists ans)
          (if (null? cars+ans) ans
              (lp cdrs (apply kons cars+ans))))))

      (let lp ((lis lis1) (ans knil))
        (if (null-list? lis) ans
            (lp (cdr lis) (kons (car lis) ans))))))
```

常见的用法：

从初始值开始，依次累加列表中的元素，返回一个数；当列表为空列表时，返回初始值。

tests/goldfish/liii/list-test.scm △ 26 ▾

```
(check (fold + 0 '(1 2 3 4)) => 10)
(check (fold + 0 '()) => 0)

(check-catch 'type-error (fold 0 + '(1 2 3 4)))
```

反转列表中的元素，返回一个新列表。

tests/goldfish/liii/list-test.scm △ 27 ▾

```
(check (fold cons () '(1 2 3 4)) => '(4 3 2 1))
```

统计列表中满足谓词的元素数量，返回这个数量。

tests/goldfish/liii/list-test.scm △ 28 ▾

```
(check
  (fold (lambda (x count) (if (symbol? x) (+ count 1) count))
        0
        '(a b 1 2 3 4))
=> 2)
```

fold-right

索引

`fold-right`与`fold`类似，不同的是，`fold-right`是从右到左折叠。

goldfish/srfi/srfi-1.scm △ 25 ▾

```
(define (fold-right f initial l)
  (if (null? l)
      initial
      (f (car l)
          (fold-right f
                      initial
                      (cdr l))))))
```

这是SRFI-1官方提供的实现，我们暂时不用：

```
(define (fold-right kons knil lis1 . lists)
```

```
(if (pair? lists)
    (let recur ((lists (cons lis1 lists)))
      (let ((cdrs (%cdrs lists)))
        (if (null? cdrs) knil
            (apply kons (%cars+ lists (recur cdrs))))))

    (let recur ((lis lis1))
      (if (null-list? lis) knil
          (let ((head (car lis)))
            (kons head (recur (cdr lis)))))))
```

在用作累加、统计时，`fold-right`与`fold`的结果是相同的。

tests/goldfish/liii/list-test.scm

△ 29 ▽

```
(check (fold-right + 0 '(1 2 3 4)) => 10)

(check (fold-right + 0 '()) => 0)

(check
  (fold-right (lambda (x count) (if (symbol? x) (+ count 1) count))
    0
    '(a b 1 2 3 4))
=>
2)
```

但`fold-right`与`fold`的折叠方向是相反的，这就使得列表原本的顺序得以保持，不会反转。

tests/goldfish/liii/list-test.scm

△ 30 ▽

```
(check (fold-right cons () '(1 2 3 4)) => '(1 2 3 4))
```

reduce

索引

`reduce`与`fold`类似，但有微妙且关键的不同。只有在列表为空列表时，才会使用这个初始值。在列表不是空列表时，则把列表的`<car>`部分取出作为`fold`的初始值，又把列表的`<cdr>`部分取出作为`fold`的列表。

goldfish/srfi/srfi-1.scm

△ 26 ▽

```
(define (reduce f initial l)
  (if (null-list? l) initial
      (fold f (car l) (cdr l))))
```

在用作累加时，`reduce`与`fold`的结果是相同的。

tests/goldfish/liii/list-test.scm

△ 31 ▽

```
(check (reduce + 0 '(1 2 3 4)) => 10)
(check (reduce + 0 '()) => 0)
```

不适用于反转列表中的元素，但当列表非空，返回的不再是列表，而是序对。因为`reduce`会把非空列表的第一个元素取出来作为`fold`的初始值。

tests/goldfish/liii/list-test.scm

△ 32 ▽

```
(check (reduce cons () '(1 2 3 4)) => '(4 3 2 . 1))
```

不适用于统计列表中满足谓词的元素数量，因为 `reduce` 会把非空列表的第一个元素取出来作为 `fold` 的初始值，引发错误。

tests/goldfish/liii/list-test.scm

△ 33 ▽

```
(check-catch 'wrong-type-arg
  (reduce (lambda (x count) (if (symbol? x) (+ count 1) count))
    0
    '(a b 1 2 3 4)))
```

reduce-right

索引

`reduce-right` 与 `reduce` 类似，不同的是，`reduce-right` 是从右到左规约。

goldfish/srfi/srfi-1.scm

△ 27 ▽

```
(define (reduce-right f initial l)
  (if (null-list? l) initial
      (let recur ((head (car l)) (l (cdr l)))
        (if (pair? l)
            (f head (recur (car l) (cdr l)))
            head))))
```

在用作累加时，`reduce-right` 与 `fold` 的结果是相同的。

tests/goldfish/liii/list-test.scm

△ 34 ▽

```
(check (reduce-right + 0 '(1 2 3 4)) => 10)

(check (reduce-right + 0 '()) => 0)
```

也不适用于重列列表中的元素，以及统计列表中满足谓词的元素数量。

tests/goldfish/liii/list-test.scm

△ 35 ▽

```
(check (reduce-right cons () '(1 2 3 4))
  => '(1 2 3 . 4) )

(check
  (reduce-right (lambda (x count) (if (symbol? x) (+ count 1) count))
    0
    '(a b 1 2 3 4))
  => 6)
```

append-map

索引

goldfish/srfi/srfi-1.scm

△ 28 ▽

```
(define append-map
  (typed-lambda ((proc procedure?) (lst list?))
    (let loop ((rest lst)
              (result '()))
      (if (null? rest)
          result
          (loop (cdr rest)
                (append result (proc (car rest)))))))
```


8.4.6 过滤和分组

filter

`filter`是一个高阶函数，接收一个谓词和一个列表为参数，从这个列表中筛出满足谓词的元素，组成一个新列表，返回这个新列表。

goldfish/srfi/srfi-1.scm

△ 29 ▾

```
(define (filter pred l)
  (let recur ((l l))
    (if (null-list? l) l
        (let ((head (car l))
              (tail (cdr l)))
          (if (pred head)
              (let ((new-tail (recur tail)))
                (if (eq? tail new-tail) l
                    (cons head new-tail)))
              (recur tail))))))
```

tests/goldfish/liii/list-test.scm

△ 36 ▾

```
(check (filter even? '(-2 -1 0 1 2)) => '(-2 0 2))
```

partition

`partition`是一个高阶函数，接收一个谓词和一个列表为参数，从这个列表中分别筛出满足和不满足谓词的元素，各组成一个新列表，返回以这两个新列表组成的序对。

goldfish/srfi/srfi-1.scm

△ 30 ▾

```
(define (partition pred l)
  (let loop ((lst l) (satisfies '()) (dissatisfies '()))
    (cond ((null? lst)
           (cons satisfies dissatisfies))
          ((pred (car lst))
           (loop (cdr lst) (cons (car lst) satisfies) dissatisfies))
          (else
           (loop (cdr lst) satisfies (cons (car lst) dissatisfies))))))
```

tests/goldfish/liii/list-test.scm

△ 37 ▾

```
(check
 (partition symbol? '(one 2 3 four five 6))
 => (cons '(five four one) '(6 3 2)))
```

remove

`remove`是一个高阶函数，接收一个谓词和一个列表为参数，从这个列表中去掉满足谓词的元素，组成一个新列表，返回这个新列表。

goldfish/srfi/srfi-1.scm

△ 31 ▾

```
(define (remove pred l)
  (filter (lambda (x) (not (pred x))) l))
```

tests/goldfish/liii/list-test.scm

△ 38 ▾

```
(check (remove even? '(-2 -1 0 1 2)) => '(-1 1))
```

8.4.7 搜索

memq

`memq`是一个S7内置函数，和`member`类似，但判断元素等价的谓词是`eq?`。也就是说，检查的是两个元素在是否是同一个实例，即它们是否具有相同的内存地址。

对比使用`equal?`的`member`或使用`eqv?`的`memv`，这种检查最为“严格”，适用于判断的元素类型最少，速度最快，适用于判断布尔值（`#t`、`#f`）、符号、整数（浮点数和复数不行）、函数（的值）这些类型的元素是否在列表中。

tests/goldfish/liii/base-test.scm

△ 67 ▽

```
(check (memq #f '(1 #f 2 3)) => '(#f 2 3))
(check (memq 'a '(1 a 2 3)) => '(a 2 3))
(check (memq 2 '(1 2 3)) => '(2 3))

(check (memq 2.0 '(1 2.0 3)) => #f)
(check (memq 2+0i '(1 2+0i 3)) => #f)

(define num1 3)
(define num2 2)
(check (memq num1 '(3 num2)) => '(3 num2))
(check (memq 3 '(num1 num2)) => #f)
(check (memq 'num1 '(num1 num2)) => '(num1 num2))

(check (memq (+ 1 1) '(1 2 3)) => '(2 3))
```

memv

`memv`是一个S7内置函数，和`member`类似，但判断元素元素的谓词是`eqv?`。也就是说，检查的是两个元素是否相同或在数值上等价。

比使用`equal?`的`member`“严格”，但比使用`eq?`的`memq`“宽松”，适用于判断的元素类型次全（`memv`能实现的功能`member`都能实现），速度中等，适用于判断数值类元素（整数、浮点数、复数）是否在列表中。但是注意，即使数值相同也不视为同一元素。

tests/goldfish/liii/base-test.scm

△ 68 ▽

```
(check (memv 2 '(1 2 3)) => '(2 3))
(check (memv 2.0 '(1 2.0 3)) => '(2.0 3))
(check (memv 2+0i '(1 2+0i 3)) => '(2+0i 3))

(check (memv 2 '(1 2.0 3)) => #f)
(check (memv 2 '(1 2+0i 3)) => #f)
```

member

`member`是一个S7内置函数，接收一个元素和一个列表为参数，返回包含该元素的第一个子列表。当元素不在列表中，返回`#f`。当列表为空列表，返回`#f`。

tests/goldfish/liii/base-test.scm

△ 69 ▽

```
(check (member 2 '(1 2 3)) => '(2 3))
(check (member 0 '(1 2 3)) => #f)
(check (member 0 '()) => #f)
```

注意，判断两个元素等价的谓词是`equal?`。也就是说，检查的是两个元素在结构和内容上是否等

价。

对比使用`eq?`的`memq`或使用`eqv?`的`memv`，这种检查最为“宽松”，适用判断的元素类型最全，但速度最慢（因为它会递归地比较复合数据结构的每个部分），建议用于判断字符串、序对、列表这些类型的元素是否在列表中，如果无需判断这些类型，建议选用`memv`或`memq`。

tests/goldfish/liii/base-test.scm

△ 70 ▽

```
(check (member "1" '(0 "1" 2 3)) => '("1" 2 3))
(check (member '(1 . 2) '(0 (1 . 2) 3)) => '((1 . 2) 3))
(check (member '(1 2) '(0 (1 2) 3)) => '((1 2) 3))
```

find

索引

`find`是一个高阶函数，接收一个谓词和一个列表为参数，返回该列表中第一个满足谓词的元素。当列表为空列表，或列表中没有满足谓词的元素，返回`#f`。

goldfish/srfi/srfi-1.scm

△ 32 ▽

```
(define (find pred l)
  (cond ((null? l) #f)
        ((pred (car l)) (car l))
        (else (find pred (cdr l)))))
```

tests/goldfish/liii/list-test.scm

△ 39 ▽

```
(check (find even? '(3 1 4 1 5 9)) => 4)

(check (find even? '()) => #f)

(check (find even? '(1 3 5 7 9)) => #f)
```

take-while

索引

`take-while`是一个高阶函数，接收一个谓词和一个列表为参数，按列表顺序筛出满足谓词的元素，直到不满足谓词的那个一个就停止筛选，返回筛出的元素组成的列表。

goldfish/srfi/srfi-1.scm

△ 33 ▽

```
(define (take-while pred lst)
  (if (null? lst)
      '()
      (if (pred (car lst))
          (cons (car lst) (take-while pred (cdr lst)))
          '()))))
```

当参数的列表为空列表，无论谓词是什么都返回空列表。

tests/goldfish/liii/list-test.scm

△ 40 ▽

```
(check (take-while even? '()) => '())
```

当列表中所有元素都满足谓词，返回原列表。

tests/goldfish/liii/list-test.scm

△ 41 ▽

```
(check (take-while (lambda (x) #t) '(1 2 3))
=> '(1 2 3))
```

当列表中没有元素满足谓词，返回空列表。

tests/goldfish/liii/list-test.scm

△ 42 ▾

```
(check
  (take-while (lambda (x) #f) '(1 2 3))
  => '())
```

当列表的第一个元素就不满足谓词，返回空列表。

tests/goldfish/liii/list-test.scm

△ 43 ▾

```
(check
  (take-while (lambda (x) (not (= x 1))) '(1 2 3))
  => '())
```

筛出元素的过程按照列表的顺序进行，当一个元素已经不满足谓词，那么这个元素之后的元素不会被筛出。

tests/goldfish/liii/list-test.scm

△ 44 ▾

```
(check
  (take-while (lambda (x) (< x 3)) '(1 2 3 0))
  => '(1 2))
```

drop-while

索引

`drop-while`是一个高阶函数，接收一个谓词和一个列表为参数，按列表顺序丢掉满足谓词的元素，直到不满足谓词的那个一个就停止丢掉，返回剩下的元素组成的列表。

goldfish/srfi/srfi-1.scm

△ 34 ▾

```
(define (drop-while pred l)
  (if (null? l)
      '()
      (if (pred (car l))
          (drop-while pred (cdr l))
          l)))
```

当列表为空列表，返回空列表。

tests/goldfish/liii/list-test.scm

△ 45 ▾

```
(check (drop-while even? '()) => '())
```

当列表中所有元素都满足谓词，返回空列表。

tests/goldfish/liii/list-test.scm

△ 46 ▾

```
(check (drop-while (lambda (x) #t) '(1 2 3)) => '())
```

当列表中没有元素满足谓词，返回原列表。

tests/goldfish/liii/list-test.scm

△ 47 ▾

```
(check (drop-while (lambda (x) #f) '(1 2 3)) => '(1 2 3))
```

当列表的第一个元素就不满足谓词，返回原列表。

tests/goldfish/liii/list-test.scm

△ 48 ▾

```
(check
  (drop-while (lambda (x) (not (= x 1))) '(1 2 3))
  => '(1 2 3))
```

list-index

索引

`list-index`是一个高阶函数，接收一个谓词和一个列表为参数，返回第一个符合谓词要求的元素的位置索引。当列表为空列表，或列表中没有满足谓词的元素，返回`#f`。

goldfish/srfi/srfi-1.scm

△ 35 ▾

```
(define (list-index pred l)
  (let loop ((index 0) (l l))
    (if (null? l)
        #f
        (if (pred (car l))
            index
            (loop (+ index 1) (cdr l))))))
```

tests/goldfish/liii/list-test.scm

△ 49 ▾

```
(check (list-index even? '(3 1 4 1 5 9)) => 2)
(check (list-index even? '()) => #f)
(check (list-index even? '(1 3 5 7 9)) => #f)
```

any

goldfish/srfi/srfi-1.scm

△ 36 ▾

```
(define (any pred? l)
  (cond ((null? l) #f)
        ((pred? (car l)) #t)
        (else (any pred? (cdr l)))))
```

tests/goldfish/liii/list-test.scm

△ 50 ▾

```
(check (any integer? '()) => #f)
(check (any integer? '(a 3.14 "3")) => #f)
(check (any integer? '(a 3.14 3)) => #t)
```

every

goldfish/srfi/srfi-1.scm

△ 37 ▾

```
(define (every pred? l)
  (cond ((null? l) #t)
        ((not (pred? (car l))) #f)
        (else (every pred? (cdr l)))))
```

tests/goldfish/liii/list-test.scm

△ 51 ▾

```
(check (every integer? '()) => #t)
(check (every integer? '(a 3.14 3)) => #f)
(check (every integer? '(1 2 3)) => #t)
```

8.4.8 删除

公共子函数，用于处理可选的`maybe-equal`参数。

goldfish/srfi/srfi-1.scm

△ 38 ▾

```
(define (%extract-maybe-equal maybe-equal)
  (let ((my-equal (if (null-list? maybe-equal)
                      =
                      (car maybe-equal))))
    (if (procedure? my-equal)
        my-equal
        (error 'wrong-type-arg "maybe-equal must be procedure"))))
```

delete

索引

goldfish/srfi/srfi-1.scm

△ 39 ▽

```
(define (delete x l . maybe-equal)
  (let ((my-equal (%extract-maybe-equal maybe-equal)))
    (filter (lambda (y) (not (my-equal x y)) l)))
```

测试用例

tests/goldfish/liii/list-test.scm

△ 52 ▽

```
(check (delete 1 (list 1 2 3 4)) => (list 2 3 4))

(check (delete 0 (list 1 2 3 4)) => (list 1 2 3 4))

(check (delete #\a (list #\a #\b #\c) char=?)
=> (list #\b #\c))

(check (delete #\a (list #\a #\b #\c) (lambda (x y) #f))
=> (list #\a #\b #\c))

(check (delete 1 (list )) => (list ))

(check
  (catch 'wrong-type-arg
    (lambda ()
      (check (delete 1 (list 1 2 3 4) 'not-pred) => 1))
    (lambda args #t))
=> #t)
```

delete-duplicates

索引

goldfish/srfi/srfi-1.scm

△ 40 ▽

```
;;; right-duplicate deletion
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; delete-duplicates delete-duplicates!
;;;
;;; Beware -- these are N^2 algorithms. To efficiently remove duplicates
;;; in long lists, sort the list to bring duplicates together, then use a
;;; linear-time algorithm to kill the dups. Or use an algorithm based on
;;; element-marking. The former gives you O(n lg n), the latter is linear.

(define (delete-duplicates lis . maybe-equal)
  (let ((my-equal (%extract-maybe-equal maybe-equal)))
    (let recur ((lis lis))
      (if (null-list? lis)
          lis
          (let* ((x (car lis))
                  (tail (cdr lis))
                  (new-tail (recur (delete x tail my-equal))))
            (if (eq? tail new-tail)
                lis
                (cons x new-tail)))))))
```

测试用例

tests/goldfish/liii/list-test.scm

△ 53 ▽

```
(check (delete-duplicates (list 1 1 2 3)) => (list 1 2 3))
(check (delete-duplicates (list 1 2 3)) => (list 1 2 3))
(check (delete-duplicates (list 1 1 1)) => (list 1))

(check (delete-duplicates (list )) => (list ))

(check (delete-duplicates (list 1 1 2 3) (lambda (x y) #f))
      => (list 1 1 2 3))
```

如果判断相等的函数类型不正确，会报错：

tests/goldfish/liii/list-test.scm

△ 54 ▽

```
(check
  (catch 'wrong-type-arg
    (lambda
      ()
      (check (delete-duplicates (list 1 1 2 3) 'not-pred) => 1))
    (lambda args #t))
  => #t)
```

8.5 三鲤扩展库

`length=?` ^{索引} `(x l) => boolean`

x. 期望的列表长度，如果长度为负数，该函数会抛出 `value-error`

1. 列表

快速判断一个列表l的长度是否为x。由于 `(= x (length l))` 这种判断方式的复杂度是 $O(n)$ ，故而需要 `length=?` 这种快速的判断方式。

tests/goldfish/liii/list-test.scm

△ 55 ▽

```
(check-true (length=? 3 (list 1 2 3)))
(check-false (length=? 2 (list 1 2 3)))
(check-false (length=? 4 (list 1 2 3)))

(check-true (length=? 0 (list )))
(check-catch 'value-error (length=? -1 (list )))
```

goldfish/liii/list.scm

△ 3 ▽

```
(define (length=? x scheme-list)
  (when (< x 0)
    (value-error "length=: expected non-negative integer x but received ~d" x))
  (cond ((and (= x 0) (null? scheme-list)) #t)
        ((or (= x 0) (null? scheme-list)) #f)
        (else (length=? (- x 1) (cdr scheme-list)))))
```

`length>=?` ^{索引} `(lst len) => bool`

使用贪心策略，先访问列表的前len个元素，如果列表长度不大于len，那么返回布尔值假，否则返回布尔值真。lst并不一定是严格意义上的列表（最后一个元素是空列表），也可能是序对。

goldfish/liii/list.scm

△ 4 ▽

```
(define (length>? lst len)
  (let loop ((lst lst)
             (cnt 0))
    (cond ((null? lst) (< len cnt))
          ((pair? lst) (loop (cdr lst) (+ cnt 1)))
          (else (< len cnt)))))
```


tests/goldfish/liii/list-test.scm

△ 56 ▽

```
(check-true (length>? '(1 2 3 4 5) 3))
(check-false (length>? '(1 2) 3))
(check-false (length>? '() 0))

(check-true (length>? '(1) 0))
(check-false (length>? '() 1))

(check-false (length>? '(1 2 . 3) 2))
(check-true (length>? '(1 2 . 3) 1))
```

length>=? (lst len) => bool 

使用贪心策略，先访问列表的前len个元素，如果列表长度小于len，那么返回布尔值假，否则返回布尔值真。lst并不一定是严格意义上的列表（最后一个元素是空列表），也可能是序对。

goldfish/liii/list.scm

△ 5 ▽

```
(define (length>=? lst len)
  (let loop ((lst lst)
             (cnt 0))
    (cond ((null? lst) (<= len cnt))
          ((pair? lst) (loop (cdr lst) (+ cnt 1)))
          (else (<= len cnt)))))
```


tests/goldfish/liii/list-test.scm

△ 57 ▽

```
(check-true (length>=? '(1 2 3 4 5) 3))
(check-false (length>=? '(1 2) 3))
(check-true (length>=? '() 0))

(check-true (length>=? '(1) 0))
(check-false (length>=? '() 1))

(check-false (length>=? '(1 2 . 3) 3))
(check-true (length>=? '(1 2 . 3) 2))
```

list-view 

由于Scheme的List和数据的流向是相反的：

```
(map (lambda (x) (* x x))
     (map (lambda (x) (+ x 1))
          (list 1 2 3)))
```


所以我们实现了 `list-view`，采用和Scala的List类似的语法来处理数据：

tests/goldfish/liii/list-test.scm△ 58 ▽

```
(check ((list-view (list 1 2 3))) => (list 1 2 3))

(check (((list-view (list 1 2 3))
  map (lambda (x) (+ x 1)))) => (list 2 3 4))

(check (((list-view (list 1 2 3))
  map (lambda (x) (+ x 1))
  map (lambda (x) (* x x))))
=> (list 4 9 16))
```

`(list-view 1 2 3)` 得到的是函数，需要在外面再加一层括号才能得到 `(list 1 2 3)`。

```
(map (lambda (x) (* x x))
  (map (lambda (x) (+ x 1))
    (list 1 2 3)))
((list-view 1 2 3)
 map (lambda (x) (+ x 1))
 map (lambda (x) (* x x)))
```

图 8.1. 使用list处理数据和使用list-view处理数据的对比

实现list-view时需要考虑三种情况和一种例外情况。

无参数. 也就是直接在list-view得到的结果外面添加括号，此时得到的是list-view对应的list

有两个参数. 这里举例说明，`((list-view 1 2 3) map (lambda (x) (+ x 1)))` 实际的计算过程是：

- 1. 计算并得到结果 `(map (lambda (x) (+ x 1)) (list 1 2 3)) => (list 2 3 4)`
- 2. 将计算结果包装到 `list-view` 里面，这里使用了 `apply` 这个内置函数

其实也是树的转换：

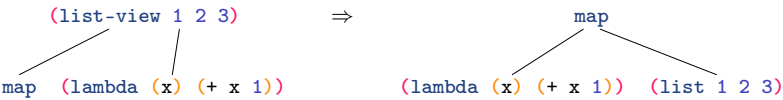


图 8.2. 原理的可视化

偶数个参数. 在上述两个递归退出条件写好的情况下，在思考这种一般的情况。

需要计算 `((list-view 1 2 3) hf1 f1 hf2 f2 ... hfn fn)`，其中hf指的是high- order function，也就是高阶函数。也就是需要计算：

```
((list-view 1 2 3) hf1 f1 hf2 f2 ... hfn fn)
```

goldfish/liii/list.scm

△ 6 ▽

```

(define (list-view scheme-list)
  (define (f-inner-reducer scheme-list filter filter-func rest-funcs)
    (cond ((null? rest-funcs) (list-view (filter filter-func scheme-list)))
          (else
           (f-inner-reducer (filter filter-func scheme-list)
                             (car rest-funcs)
                             (cadr rest-funcs)
                             (cddr rest-funcs)))))
  (define (f-inner . funcs)
    (cond ((null? funcs) scheme-list)
          ((length=? 2 funcs)
           (list-view ((car funcs) (cadr funcs) scheme-list)))
          ((even? (length funcs))
           (f-inner-reducer scheme-list
                             (car funcs)
                             (cadr funcs)
                             (cddr funcs)))
          (else (error 'wrong-number-of-args
                        "list-view only accepts even number of args"))))
  f-inner)

```

flatmap

索引

goldfish/liii/list.scm

△ 7 ▽

```

(define flatmap append-map)

```

tests/goldfish/liii/list-test.scm

△ 59 ▽

```

(check (flatmap (lambda (x) (list x x))
                (list 1 2 3))
      => (list 1 1 2 2 3 3))

(check-catch 'type-error (flatmap 1 (list 1 2 3)))

```

not-null-list?

索引

null-list?的反面，会抛出异常。

goldfish/liii/list.scm

△ 8 ▽

```

(define (not-null-list? l)
  (cond ((pair? l)
        (or (null? (cdr l)) (pair? (cdr l))))
        ((null? l) #f)
        (else
         (error 'type-error "type mismatch"))))

```

list-null?

索引

null-list?的没有异常的版本，只要不是list，都是#f。

goldfish/liii/list.scm

△ 9 ▽

```

(define (list-null? l)
  (and (not (pair? l)) (null? l)))

```

list-not-null?

索引

`not-null-list?`的没有异常的版本。

goldfish/liii/list.scm

△ 10 ▽

```
(define (list-not-null? l)
  (and (pair? l)
       (or (null? (cdr l)) (pair? (cdr l)))))
```

tests/goldfish/liii/list-test.scm

△ 60 ▽

```
(check (not-null-list? (list 1)) => #t)
(check (list-not-null? (list 1)) => #t)
(check (list-null? (list 1)) => #f)

(check (not-null-list? (list 1 2 3)) => #t)
(check (list-not-null? (list 1 2 3)) => #t)
(check (list-null? (list 1 2 3)) => #f)

(check (not-null-list? '(a)) => #t)
(check (list-not-null? '(a)) => #t)
(check (list-null? '(a)) => #f)

(check (not-null-list? '(a b c)) => #t)
(check (list-not-null? '(a b c)) => #t)
(check (list-null? '(a b c)) => #f)

(check (not-null-list? ()) => #f)
(check (list-not-null? ()) => #f)
(check (list-null? ()) => #t)

; '(a) is a pair and a list
; '(a . b) is a pair but not a list
(check (not-null-list? '(a . b)) => #f)
(check (list-not-null? '(a . b)) => #f)
(check (list-null? '(a . b)) => #f)

(check-catch 'type-error (not-null-list? 1))
(check (list-not-null? 1) => #f)
(check (list-null? 1) => #f)
```

8.6 结尾

goldfish/liii/list.scm

△ 11

```
) ; end of begin
) ; end of library
```

goldfish/srfi/srfi-1.scm

△ 41

```
) ; end of begin
) ; end of define-library
```

tests/goldfish/liii/list-test.scm

△ 61

```
(check-report)
```


第 9 章

(liii vector)

chapter:liiii_vector

9.1 许可证

goldfish/srfi/srfi-133.scm1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

goldfish/liiii/vector.scm1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

[tests/goldfish/liii/vector-test.scm](#)

1 ▾

```

;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;

```

9.2 接口

[goldfish/srfi/srfi-133.scm](#)

△ 2 ▾

```

(define-library (srfi srfi-133)

  (export
    vector-empty?
    vector-count
    vector-any vector-every vector-copy vector-copy!
    vector-index vector-index-right vector-partition
    vector-swap!)
  (import (scheme base))
  (begin

```

[goldfish/liii/vector.scm](#)

△ 2 ▾

```

(define-library (liii vector)

  (export
    ; S7 Scheme built-in
    make-vector vector-length vector-ref vector-set! vector->list list->vector
    ; from (scheme base)
    vector-copy vector-fill! vector-copy! vector->string string->vector
    vector-map vector-for-each
    ; from (srfi srfi-133)
    vector-empty?
    vector-count
    vector-any vector-every vector-copy vector-copy!
    vector-index vector-index-right vector-partition
    vector-swap!)
  (import (srfi srfi-133)
    (scheme base))
  (begin

```

9.3 测试

tests/goldfish/liii/vector-test.scm

△ 2 ▽

```
(import (liii list)
        (liii check)
        (liii vector)
        (only (scheme base) let-values))

(check-set-mode! 'report-failed)

(for-each (lambda (p) (check (procedure? p) => #t))
  (list
    vector-empty?
    vector-count
    vector-any vector-every vector-copy vector-copy!
    vector-index vector-index-right vector-partition
    vector-swap!))
```

9.4 实现

9.4.1 构造器

索引 **make-vector** (k [fill]) => **vector** 索引

返回长度为k的向量，如果提供了fill，则采用fill作为每一个元素的初始值，否则每一个元素的初始值是未指定。

tests/goldfish/liii/base-test.scm

△ 71 ▽

```
(check (make-vector 1 1) => (vector 1))
(check (make-vector 3 'a) => (vector 'a 'a 'a))

(check (make-vector 0) => (vector ))
(check (vector-ref (make-vector 1) 0) => #<unspecified>)
```

索引 **vector** (obj1 obj2 ...) => **vector** 索引

返回一个新分配的向量，其元素包含给定的参数。类似于list。

tests/goldfish/liii/base-test.scm

△ 72 ▽

```
(check (vector 'a 'b 'c) => #(a b c))
(check (vector) => #())
```

索引 **int-vector** (integer integer ...) => **vector** 索引

返回一个所有元素都是integer类型的int-vector。int-vector是vector的子类型。

tests/goldfish/liii/vector-test.scm



△ 3 ▽

```
(check-true (vector? (int-vector 1 2 3)))
(check-catch 'wrong-type-arg (int-vector 1 2 'a))

(let1 v (int-vector 1 2 3)
  (check (vector-ref v 0) => 1)
  (check (vector-ref v 1) => 2)
  (check (vector-ref v 2) => 3))
```

vector-unfold

vector-unfold-right

 **vector-copy** (*v* [*start* [*end*]]) => **vector** 

返回一个新的分配副本，包含给定向量从开始到结束的元素。新向量的元素与旧向量的元素相同（在eqv?的意义上）。

```
(define (vector-copy v)
  (let ((new-v (make-vector (vector-length v))))
    (let loop ((i 0))
      (if (= i (vector-length v))
          new-v
          (begin
             (vector-set! new-v i (vector-ref v i))
             (loop (+ i 1)))))))
```

goldfish/scheme/base.scm

△ 29 ▽

```
(define* (vector-copy v (start 0) (end (vector-length v)))
  (if (or (> start end) (> end (vector-length v)))
      (error 'out-of-range "vector-copy")
      (let ((new-v (make-vector (- end start))))
        (let loop ((i start) (j 0))
          (if (>= i end)
              new-v
              (begin
                 (vector-set! new-v j (vector-ref v i))
                 (loop (+ i 1) (+ j 1)))))))
```

tests/goldfish/liii/vector-test.scm

△ 4 ▽



```
(check (vector-copy #(0 1 2 3)) => #(0 1 2 3))
(check (vector-copy #(0 1 2 3) 1) => #(1 2 3))
(check (vector-copy #(0 1 2 3) 3) => #(3))
(check (vector-copy #(0 1 2 3) 4) => #())

(check-catch 'out-of-range (vector-copy #(0 1 2 3) 5))
(check-catch 'out-of-range (vector-copy #(0 1 2 3) 1 5))

(define my-vector #(0 1 2 3))
(check (eqv? my-vector (vector-copy #(0 1 2 3))) => #f)
(check-true
  (eqv? (vector-ref my-vector 2)
        (vector-ref (vector-copy #(0 1 2 3)) 2)))

(check (vector-copy #(0 1 2 3) 1 1) => #())
(check (vector-copy #(0 1 2 3) 1 2) => #(1))
(check (vector-copy #(0 1 2 3) 1 4) => #(1 2 3))
```

vector-reverse-copy

 **vector-append** (*v1 v2 v3 ...*) => **vector** 

返回一个新分配的向量，其元素是给定向量的元素的拼接。这是一个S7内置的函数。

tests/goldfish/liii/base-test.scm

△ 73 ▾

```
(check (vector-append #(0 1 2) #(3 4 5)) => #(0 1 2 3 4 5))
```

9.4.2 谓词

vector?  (vector? obj) => bool

如果obj是一个向量返回#t, 否则返回#f。

tests/goldfish/liii/base-test.scm

△ 74 ▾

```
(check (vector? #(1 2 3)) => #t)
(check (vector? #()) => #t)
(check (vector? '(1 2 3)) => #f)
```


int-vector? (int-vector? obj) => bool

只有使用int-vector构造的vector, 才能在判定为真。

tests/goldfish/liii/vector-test.scm

△ 5 ▾

```
(check-true (int-vector? (int-vector 1 2 3)))
(check-false (int-vector? (vector 1 2 3)))
```

vector-empty? 

goldfish/srfi/srfi-133.scm

△ 3 ▾

```
(define (vector-empty? v)
  (when (not (vector? v))
    (error 'type-error "v is not a vector")))
(zero? (vector-length v)))
```

tests/goldfish/liii/vector-test.scm

△ 6 ▾

```
(check-true (vector-empty? (vector)))
(check-false (vector-empty? (vector 1)))
(check-catch 'type-error (vector-empty? 1))
```

vector=

9.4.3 选择器

vector-length (v) => integer 索引

以整数返回向量中元素的数量。

tests/goldfish/liii/base-test.scm

△ 75 ▽

```
(check (vector-length #(1 2 3)) => 3)
(check (vector-length #()) => 0)
```

vector-ref (vector-ref v k) 索引

返回向量中索引为 k 的元素的内容。当 k 不是向量的有效索引，报错。

tests/goldfish/liii/base-test.scm

△ 76 ▽

```
(check (vector-ref #(1 2 3) 0) => 1)
(check (vector-ref #(1 2 3) 2) => 3)

(check-catch 'out-of-range (vector-ref #(1 2 3) 3))
(check-catch 'out-of-range (vector-ref #() 0))

(check
  (catch 'wrong-type-arg
    (lambda () (vector-ref #(1 2 3) 2.0))
    (lambda args #t))
  =>
  #t)

(check
  (catch 'wrong-type-arg
    (lambda () (vector-ref #(1 2 3) "2"))
    (lambda args #t))
  =>
  #t)
```

9.4.4 迭代

vector-map 索引

goldfish/scheme/base.scm

△ 30 ▽

```
(define (vector-map p . args) (apply vector (apply map p args)))
```

vector-for-each (proc vector1 [vector2 ...]) 索引

实现

goldfish/scheme/base.scm

△ 31 ▽

```
(define vector-for-each for-each)
```

测试

tests/goldfish/liii/vector-test.scm

△ 7 ▽

```
(check
  (let ((lst (make-list 5)))
    (vector-for-each
      (lambda (i) (list-set! lst i (* i i)))
      #(0 1 2 3 4))
    lst)
  => '(0 1 4 9 16))

(check
  (let ((lst (make-list 5)))
    (vector-for-each
      (lambda (i) (list-set! lst i (* i i)))
      #(0 1 2))
    lst)
  => '(0 1 4 #f #f))

(check
  (let ((lst (make-list 5)))
    (vector-for-each
      (lambda (i) (list-set! lst i (* i i)))
      #())
    lst)
  => '(#f #f #f #f #f))
```

vector-count

索引

实现

goldfish/srfi/srfi-133.scm

△ 4 ▽

```
; TODO optional parameters
(define (vector-count pred v)
  (let loop ((i 0) (count 0))
    (cond ((= i (vector-length v)) count)
          ((pred (vector-ref v i))
           (loop (+ i 1) (+ count 1)))
          (else (loop (+ i 1) count)))))
```

测试

tests/goldfish/liii/vector-test.scm

△ 8 ▽

```
(check (vector-count even? #()) => 0)
(check (vector-count even? #(1 3 5 7 9)) => 0)
(check (vector-count even? #(1 3 4 7 8)) => 2)
```

9.4.5 搜索

vector-any

索引

实现

goldfish/srfi/srfi-133.scm

△ 5 ▽

```
; TODO optional parameters
(define (vector-any pred v)
  (let loop ((i 0))
    (cond ((= i (vector-length v)) #f)
          ((pred (vector-ref v i)) #t)
          (else (loop (+ i 1))))))
```

测试

tests/goldfish/liii/vector-test.scm

△ 9 ▽

```
(check (vector-any even? #()) => #f)
(check (vector-any even? #(1 3 5 7 9)) => #f)
(check (vector-any even? #(1 3 4 7 8)) => #t)
```

vector-every

索引

goldfish/srfi/srfi-133.scm

△ 6 ▽

```
; TODO optional parameters
(define (vector-every pred v)
  (let loop ((i 0))
    (cond ((= i (vector-length v)) #t)
          ((not (pred (vector-ref v i))) #f)
          (else (loop (+ i 1))))))
```

tests/goldfish/liii/vector-test.scm

△ 10 ▽

```
(check (vector-every odd? #()) => #t)
(check (vector-every odd? #(1 3 5 7 9)) => #t)
(check (vector-every odd? #(1 3 4 7 8)) => #f)
```

vector-index

索引

goldfish/srfi/srfi-133.scm

△ 7 ▽

```
; TODO optional parameters
(define (vector-index pred v)
  (let loop ((i 0))
    (cond ((= i (vector-length v)) #f)
          ((pred (vector-ref v i)) i)
          (else (loop (+ i 1))))))
```

tests/goldfish/liii/vector-test.scm

△ 11 ▽

```
(check (vector-index even? #()) => #f)
(check (vector-index even? #(1 3 5 7 9)) => #f)
(check (vector-index even? #(1 3 4 7 8)) => 2)
```

vector-index-right

索引

goldfish/srfi/srfi-133.scm

△ 8 ▽

```
; TODO optional parameters
(define (vector-index-right pred v)
  (let ((len (vector-length v)))
    (let loop ((i (- len 1)))
      (cond ((< i 0) #f)
            ((pred (vector-ref v i)) i)
            (else (loop (- i 1)))))))
```

tests/goldfish/liii/vector-test.scm

△ 12 ▾

```
(check (vector-index-right even? #()) => #f)
(check (vector-index-right even? #(1 3 5 7 9)) => #f)
(check (vector-index-right even? #(1 3 4 7 8)) => 4)
```

vector-partition

索引

goldfish/srfi/srfi-133.scm

△ 9 ▾

```
(define (vector-partition pred v)
  (let* ((len (vector-length v))
        (cnt (vector-count pred v))
        (ret (make-vector len)))
    (let loop ((i 0) (yes 0) (no cnt))
      (if (= i len)
          (values ret cnt)
          (let ((elem (vector-ref v i)))
            (if (pred elem)
                (begin
                  (vector-set! ret yes elem)
                  (loop (+ i 1) (+ yes 1) no))
                (begin
                  (vector-set! ret no elem)
                  (loop (+ i 1) yes (+ no 1))))))))))
```

tests/goldfish/liii/vector-test.scm

△ 13 ▾

```
(define (vector-partition->list pred v)
  (let-values (((ret cnt) (vector-partition pred v))) (list ret cnt)))

(check (vector-partition->list even? #()) => '(#() 0))
(check (vector-partition->list even? #(1 3 5 7 9)) => '(#(1 3 5 7 9) 0))
(check (vector-partition->list even? #(1 3 4 7 8)) => '(#(4 8 1 3 7) 2))
```

9.4.6 修改器

r7rs

vector-set! (vector-set! v k obj)

索引

该函数将对象 obj 存储到向量中索引为 k 的元素里。注意，返回的不是向量，而是那个 obj。当 k 不是向量的有效索引，报错。

tests/goldfish/liii/base-test.scm

△ 77 ▾

```
(define my-vector #(0 1 2 3))
(check my-vector => #(0 1 2 3))

(check (vector-set! my-vector 2 10) => 10)
(check my-vector => #(0 1 10 3))

(check
  (catch 'out-of-range
    (lambda () (vector-set! my-vector 4 10))
    (lambda (args #t))
  =>
  #t)
```

vector-swap!

索引

goldfish/srfi/srfi-133.scm

△ 10 ▽

```
(define (vector-swap! vec i j)
  (let ((elem-i (vector-ref vec i))
        (elem-j (vector-ref vec j)))
    (vector-set! vec i elem-j)
    (vector-set! vec j elem-i)
  ))
```

tests/goldfish/liii/vector-test.scm

△ 14 ▽

```
(define my-vector (vector 0 1 2 3))
(vector-swap! my-vector 1 2)
(check my-vector => #(0 2 1 3))

(define my-vector (vector 0 1 2 3))
(vector-swap! my-vector 1 1)
(check my-vector => #(0 1 2 3))

(define my-vector (vector 0 1 2 3))
(vector-swap! my-vector 0 (- (vector-length my-vector) 1))
(check my-vector => #(3 1 2 0))

(check-catch
  'out-of-range
  (vector-swap! my-vector 1 (vector-length my-vector)))
```

x7f88

vector-fill!

索引

goldfish/scheme/base.scm

△ 32 ▽

```
(define vector-fill! fill!)
```

tests/goldfish/liii/vector-test.scm

△ 15 ▽

```
(define my-vector (vector 0 1 2 3 4))
(fill! my-vector #f)
(check my-vector => #( #f #f #f #f #f))

(define my-vector (vector 0 1 2 3 4))
(fill! my-vector #f 1 2)
(check my-vector => #(0 #f 2 3 4))
```

x7f88

vector-copy! (vector-copy! to at from [start [end]])

索引

to和from都是向量，**vector-copy!**就是从from向量复制元素依次粘贴到to向量，粘贴从to向量的第at个索引位置开始；start和end是可选参数，用于指定从from向量选取元素的范围。

```
a (vector "a0" "a1" "a2" "a3" "a4")
b (vector "b0" "b1" "b2" "b3" "b4")
```

(vector-copy! b 1 a 0 2) 就是：

第一步：从a中选取索引为[0,2)的元素，即"a0" "a1"

第二步：定位到b的索引为 1 的位置，即"b1"所在的那个位置

第二步：以刚刚定位到的那个"b1"位置为起点，把刚刚选出的元素"a0" "a1"依次替换，直到

代码实现时要注意

at、to、from 的边界条件，按顺序写它们会导致报错的条件
要注意每一个参数在后面会遇到怎样的使用

第一步：只涉及单个参数的报错条件

```
(< at 0) 可省去
(> at (vector-length to)) 可省去
(< start 0)
(< start 0)
(> start (vector-length from))
(< end 0)
(> end (vector-length from))
```

第二步：两个依赖关系的参数的报错条件

```
(> start end)
```

第三步：三个依赖关系参数的报错条件

```
(> (+ at (- end start)) (vector-length to))
```

综合上方的报错条件，去掉一些多余的条件：

```
(< start 0) 可省去
因有 (> (+ at (- end start)) (vector-length to))，所以 (> at (vector-length to)) 可省去
```

```
> (define a (vector "a0" "a1" "a2" "a3" "a4"))
> (define b (vector "b0" "b1" "b2" "b3" "b4"))
> (vector-copy! b 0 a 1)
> b
```

```
#("a1" "a2" "a3" "a4" "b4")
```

```
> a
>
```

goldfish/scheme/base.scm

△ 33 ▽

```
(define* (vector-copy! to at from (start 0) (end (vector-length from)))
  (if (or (< at 0)
        (> start (vector-length from))
        (< end 0)
        (> end (vector-length from))
        (> start end)
        (> (+ at (- end start)) (vector-length to)))
      (error 'out-of-range "vector-copy!")
      (let loop ((to-i at) (from-i start))
        (if (>= from-i end)
            to
            (begin
              (vector-set! to to-i (vector-ref from from-i))
              (loop (+ to-i 1) (+ from-i 1)))))))
```

tests/goldfish/liii/vector-test.scm

△ 16 ▽

```

(define a (vector "a0" "a1" "a2" "a3" "a4"))
(define b (vector "b0" "b1" "b2" "b3" "b4"))

;(< at 0)
(check
  (catch 'out-of-range
    (lambda () (vector-copy! b -1 a))
    (lambda args #t)))
=>
#t)

;(< start 0)
(check
  (catch 'out-of-range
    (lambda () (vector-copy! b 0 a -1))
    (lambda args #t)))
=>
#t)

;(> start (vector-length from))
(check
  (catch 'out-of-range
    (lambda () (vector-copy! b 0 a 6))
    (lambda args #t)))
=>
#t)

;(> end (vector-length from))
(check
  (catch 'out-of-range
    (lambda () (vector-copy! b 0 a 0 6))
    (lambda args #t)))
=>
#t)

;(> start end)
(check
  (catch 'out-of-range
    (lambda () (vector-copy! b 0 a 2 1))
    (lambda args #t)))
=>
#t)

;(> (+ at (- end start)) (vector-length to))
(check
  (catch 'out-of-range
    (lambda () (vector-copy! b 6 a))
    (lambda args #t)))
=>
#t)

(check
  (catch 'out-of-range
    (lambda () (vector-copy! b 1 a))
    (lambda args #t)))
=>
#t)

```


9.4.7 转换

`r7rs` `vector->list` (vector->list v [start [end]]) `索引`

返回一个新分配的列表，包含向量中从 start 到 end 之间的元素中的对象。当 start、end 不是向量的有效索引，报错。

tests/goldfish/liii/base-test.scm

△ 78 ▾

```
(check (vector->list #()) => '())
(check (vector->list #() 0) => '())

(check
  (catch 'out-of-range
    (lambda () (vector->list #() 1))
    (lambda args #t))
  =>
  #t)

(check (vector->list #(0 1 2 3)) => '(0 1 2 3))
(check (vector->list #(0 1 2 3) 1) => '(1 2 3))
(check (vector->list #(0 1 2 3) 1 1) => '())
(check (vector->list #(0 1 2 3) 1 2) => '(1))
```

`r7rs` `list->vector` (list->vector l) `索引`

返回一个新创建的向量，其元素初始化为列表 list 中的元素。注意，`list->vector` 不像 `vector->list` 那样可用接收索引参数。

tests/goldfish/liii/base-test.scm

△ 79

```
(check (list->vector '(0 1 2 3)) => #(0 1 2 3))
(check (list->vector '()) => #())
```

`r7rs` `vector->string` (v [start [end]]) => string `索引`

将向量 v 转化为字符串，如果指定了起始索引和终止索引，则只将指定范围内的子向量转化为字符串。

goldfish/scheme/base.scm

△ 34 ▾

```
; 0-clause BSD
; Bill Schottstaedt
; from S7 source repo: r7rs.scm
(define* (vector->string v (start 0) end)
  (let ((stop (or end (length v))))
    (copy v (make-string (- stop start)) start stop)))
```

tests/goldfish/liii/vector-test.scm

△ 17 ▽

```

(check (vector->string (vector #\0 #\1 #\2 #\3)) => "0123")
(check (vector->string (vector #\a #\b #\c)) => "abc")



(check (vector->string (vector #\0 #\1 #\2 #\3) 0 4) => "0123")
(check (vector->string (vector #\0 #\1 #\2 #\3) 1) => "123")
(check (vector->string (vector #\0 #\1 #\2 #\3) 1 4) => "123")
(check (vector->string (vector #\0 #\1 #\2 #\3) 1 3) => "12")
(check (vector->string (vector #\0 #\1 #\2 #\3) 1 2) => "1")

(check
  (catch 'out-of-range
    (lambda () (vector->string (vector #\0 #\1 #\2 #\3) 2 10))
    (lambda args #t))
  =>
  #t)

(check (vector->string (vector 0 1 #\2 3 4) 2 3) => "2")

(check
  (catch 'wrong-type-arg
    (lambda () (vector->string (vector 0 1 #\2 3 4) 1 3))
    (lambda args #t))
  =>
  #t)

```

 **string->vector** (s [start [end]]) =>  vector

将字符串s转化为向量，如果指定了起始索引和终止索引，则只将指定范围内的子字符串转化为向量。

goldfish/scheme/base.scm

△ 35 ▽

```

; 0-clause BSD
; Bill Schottstaedt
; from S7 source repo: r7rs.scm
(define* (string->vector s (start 0) end)
  (let ((stop (or end (length s))))
    (copy s (make-vector (- stop start)) start stop)))

```

tests/goldfish/liii/vector-test.scm

△ 18 ▽

```

(check (string->vector "0123") => (vector #\0 #\1 #\2 #\3))
(check (string->vector "abc") => (vector #\a #\b #\c))

(check (string->vector "0123" 0 4) => (vector #\0 #\1 #\2 #\3))
(check (string->vector "0123" 1) => (vector #\1 #\2 #\3))
(check (string->vector "0123" 1 4) => (vector #\1 #\2 #\3))
(check (string->vector "0123" 1 3) => (vector #\1 #\2))
(check (string->vector "0123" 1 2) => (vector #\1))

(check-catch 'out-of-range (string->vector "0123" 2 10))

```

9.5 结尾

[goldfish/srfi/srfi-133.scm](#)

[△ 11](#)

```
) ; end of begin  
) ; end of define-library
```

[goldfish/liii/vector.scm](#)

[△ 3](#)

```
) ; end of begin  
) ; end of define-library
```

[tests/goldfish/liii/vector-test.scm](#)

[△ 19](#)

```
(check-report)
```

第 10 章

(liii stack)

栈是一个先进后出（FILO）的数据结构。

这个函数库是三鲤自定义的库，参考了C++和Java的栈相关的函数库的接口。

10.1 许可证

goldfish/liii/stack.scm

1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

10.2 接口

goldfish/liii/stack.scm

△ 2 ▾

```
(define-library (liii stack)
(import (srfi srfi-9)
        (liii base)
        (liii error))
(export
 stack
 stack? stack-empty?
 stack-size stack-top
 stack-push! stack-pop!
 stack->list
)
(begin
```

10.3 测试

tests/goldfish/liii/stack-test.scm

```
(import (liii stack)
        (liii check))

(check-set-mode! 'report-failed)

(define stack1 (stack))
(check (stack-empty? stack1) => #t)
(check (stack->list stack1) => '())
(check-catch 'value-error (stack-pop! stack1))

(stack-push! stack1 1)
(check (stack->list stack1) => '(1))
(check (stack-top stack1) => 1)
(check (stack-size stack1) => 1)
(check (stack-pop! stack1) => 1)
(check (stack-empty? stack1) => #t)
(check (stack-size stack1) => 0)

(define stack2 (stack 1 2 3))
(check (stack->list stack2) => '(1 2 3))
(check (stack-size stack2) => 3)
(check (stack-pop! stack2) => 1)
(check (stack-pop! stack2) => 2)
(check (stack-pop! stack2) => 3)

(define stack3 (stack ))
(stack-push! stack3 1)
(stack-push! stack3 2)
(stack-push! stack3 3)
(check (stack-pop! stack3) => 3)
(check (stack-pop! stack3) => 2)
(check (stack-pop! stack3) => 1)

(check-catch 'type-error (stack-empty? 1))
```


10.4 实现

goldfish/liii/stack.scm

△ 3 ▽

```
(define-record-type :stack
  (make-stack data)
  stack?
  (data get-data set-data!))

(define (%stack-check-parameter st)
  (when (not (stack? st))
    (error 'type-error "Parameter st is not a stack")))
```

stack  (x1 x2 ...) => stack

传入参数，构造一个栈，第一个参数是栈顶。如果没有参数，则构造的是空栈。

[goldfish/liii/stack.scm](#)[△ 4 ▾](#)

```
(define (stack . l)
  (if (null? l)
      (make-stack '())
      (make-stack l)))
```

stack-empty? ^{索引} (st) => bool

[goldfish/liii/stack.scm](#)[△ 5 ▾](#)

```
(define (stack-empty? st)
  (%stack-check-parameter st)
  (null? (get-data st)))
```

stack-size ^{索引} (st) => int

[goldfish/liii/stack.scm](#)[△ 6 ▾](#)

```
(define (stack-size st)
  (%stack-check-parameter st)
  (length (get-data st)))
```

stack-top ^{索引} (st) => x

[goldfish/liii/stack.scm](#)[△ 7 ▾](#)

```
(define (stack-top st)
  (%stack-check-parameter st)
  (car (get-data st)))
```

stack-push! ^{索引} (st x) => #<unspecified>

[goldfish/liii/stack.scm](#)[△ 8 ▾](#)

```
(define (stack-push! st elem)
  (%stack-check-parameter st)
  (set-data! st (cons elem (get-data st))))
```

stack-pop! ^{索引} (st) => x

[goldfish/liii/stack.scm](#)[△ 9 ▾](#)

```
(define (stack-pop! st)
  (%stack-check-parameter st)
  (when (stack-empty? st)
    (error 'value-error "Failed to stack-pop! on empty stack"))
  (let1 data (get-data st)
    (set-data! st (cdr data))
    (car data)))
```

stack->list ^{索引} (st) => list

[goldfish/liii/stack.scm](#)[△ 10 ▾](#)

```
(define (stack->list st)
  (%stack-check-parameter st)
  (get-data st))
```

10.5 结尾

[goldfish/liii/stack.scm](#)[△ 11](#)

```
) ; end of begin
) ; end of library
```


第 11 章

(liii queue)

这个函数库是三鲤自定义的库，参考了C++和Java的队列相关的函数库的接口。目前基于Scheme的列表实现，从队列取出数据的复杂度是 $O(1)$ ，从队列存入数据的复杂度是 $O(n)$ 。

11.1 许可证

[goldfish/liii/queue.scm](#) 1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

[tests/goldfish/liii/queue-test.scm](#) 1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

11.2 接口

goldfish/liii/queue.scm

△ 2 ▽

```
(define-library (liii queue)
  (import (liii list)
          (liii base)
          (srfi srfi-9)
          (liii error))
  (export
   queue
   queue? queue-empty?
   queue-size queue-front queue-back
   queue-pop! queue-push!
   queue->list)
  (begin
```

11.3 测试

tests/goldfish/liii/queue-test.scm

△ 2 ▽

```
(import (liii queue)
        (liii base)
        (liii check))

(check-set-mode! 'report-failed)

(let1 q1 (queue)
  (check-true (queue-empty? q1))
  (check (queue-size q1) => 0)
  (check-catch 'value-error (queue-pop! q1))

  (queue-push! q1 1)
  (check (queue-size q1) => 1)
  (check (queue-front q1) => 1)
  (check (queue-back q1) => 1)
  (check (queue-pop! q1) => 1)
  (check-true (queue-empty? q1))
)

(let1 q2 (queue 1 2 3)
  (check (queue-size q2) => 3)
  (check (queue-front q2) => 1)
  (check (queue-back q2) => 3)
  (check (queue-pop! q2) => 1)
  (check (queue-pop! q2) => 2)
  (check (queue-pop! q2) => 3)
)
```

11.4 实现

goldfish/liii/queue.scm

△ 3 ▽

```
(define-record-type :queue
  (make-queue data)
  queue?
  (data get-data set-data!))

(define (%queue-assert-type q)
  (when (not (queue? q))
    (type-error "Parameter q is not a queue"))))

(define (%queue-assert-value q)
  (when (queue-empty? q)
    (value-error "q must be non-empty")))
```

queue

索引

第一个参数是队列的头部，最后一个参数是队列的尾部。

goldfish/liii/queue.scm

△ 4 ▽

```
(define (queue . l)
  (if (null? l)
      (make-queue '())
      (make-queue l)))
```

queue-empty?

索引

(queue) => bool

goldfish/liii/queue.scm

△ 5 ▽

```
(define (queue-empty? q)
  (%queue-assert-type q)
  (null? (get-data q)))
```

queue-size

索引

(queue) => int

goldfish/liii/queue.scm

△ 6 ▽

```
(define (queue-size q)
  (%queue-assert-type q)
  (length (get-data q)))
```

queue-front

索引

(queue) => x

goldfish/liii/queue.scm

△ 7 ▽

```
(define (queue-front q)
  (%queue-assert-type q)
  (%queue-assert-value q)
  (first (get-data q)))
```

queue-back

索引

(queue) => x

goldfish/liii/queue.scm

△ 8 ▽


```
(define (queue-back q)
  (%queue-assert-type q)
  (%queue-assert-value q)
  (last (get-data q)))
```

queue-push!  `(queue x) => queue`

[goldfish/liii/queue.scm](#)

[△ 9 ▾](#)

```
(define (queue-push! q x)
  (%queue-assert-type q)
  (let1 data (get-data q)
    (set-data! q (append data (list x))))))
```

queue-pop!  `(queue) => x`

[goldfish/liii/queue.scm](#)

[△ 10 ▾](#)

```
(define (queue-pop! q)
  (%queue-assert-type q)
  (%queue-assert-value q)
  (let1 data (get-data q)
    (set-data! q (cdr data))
    (car data)))
```

queue->list  `(queue) => list`

[goldfish/liii/queue.scm](#)

[△ 11 ▾](#)

```
(define (queue->list q)
  (get-data q))
```

11.5 结尾

[goldfish/liii/queue.scm](#)

[△ 12](#)

```
) ; end of begin
) ; end of library
```

[tests/goldfish/liii/queue-test.scm](#)

[△ 3](#)

```
(check-report)
```

第 12 章

(liii comparator)

12.1 许可证

SRFI 128的参考实现依赖于SRFI 39。我们可以移除SRFI 128的参考实现对于SRFI 39的依赖。

goldfish/liii/comparator.scm1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

goldfish/srfi/srfi-128.scm

1 ▾

```

;;; SPDX-License-Identifier: MIT
;;;
;;; Copyright (C) John Cowan (2015). All Rights Reserved.
;;;
;;; Permission is hereby granted, free of charge, to any person
;;; obtaining a copy of this software and associated documentation
;;; files (the "Software"), to deal in the Software without
;;; restriction, including without limitation the rights to use,
;;; copy, modify, merge, publish, distribute, sublicense, and/or
;;; sell copies of the Software, and to permit persons to whom the
;;; Software is furnished to do so, subject to the following
;;; conditions:
;;;
;;; The above copyright notice and this permission notice shall be
;;; included in all copies or substantial portions of the Software.
;;;
;;; THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
;;; EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
;;; OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
;;; NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
;;; HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
;;; WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
;;; FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
;;; OTHER DEALINGS IN THE SOFTWARE.

;;;; Main part of the SRFI 114 reference implementation

;;; "There are two ways of constructing a software design: One way is to
;;; make it so simple that there are obviously no deficiencies, and the
;;; other way is to make it so complicated that there are no *obvious*
;;; deficiencies." --Tony Hoare

```

12.2 接口

goldfish/liii/comparator.scm

△ 2 ▾

```

(define-library (liii comparator)
  (import (srfi srfi-128))
  (export
    comparator? comparator-ordered? comparator-hashable?
    make-comparator make-pair-comparator make-list-comparator
    make-vector-comparator make-eq-comparator make-eqv-comparator make-equal-comparator
    boolean-hash char-hash char-ci-hash string-hash string-ci-hash
    symbol-hash number-hash
    make-default-comparator default-hash
    comparator-type-test-predicate comparator-equality-predicate
    comparator-ordering-predicate comparator-hash-function
    comparator-test-type comparator-check-type comparator-hash
    =? <? >? <=? >=?
  )
  (begin

```

goldfish/srfi/srfi-128.scm

△ 2 ▽

```

(define-library (srfi srfi-128)
  (import (scheme base)
          (liii error))
  (export
    comparator? comparator-ordered? comparator-hashable?
    make-comparator make-pair-comparator make-list-comparator
    make-vector-comparator make-eq-comparator make-equiv-comparator make-equal-comparator
    boolean-hash char-hash char-ci-hash string-hash string-ci-hash
    symbol-hash number-hash
    make-default-comparator default-hash
    comparator-type-test-predicate comparator-equality-predicate
    comparator-ordering-predicate comparator-hash-function
    comparator-test-type comparator-check-type comparator-hash
    =? <? >? <=? >=?
  )
  (begin

```

comparator?

索引

comparator-ordered?

索引

comparator-hashable?

索引

comparator-type-test-predicate

索引

comparator-equality-predicate

索引

comparator-ordering-predicate

索引

comparator-hash-function

索引

goldfish/srfi/srfi-128.scm

△ 3 ▽

```

(define-record-type comparator
  (make-raw-comparator type-test equality ordering hash ordering? hash?)
  comparator?
  (type-test comparator-type-test-predicate)
  (equality comparator-equality-predicate)
  (ordering comparator-ordering-predicate)
  (hash comparator-hash-function)
  (ordering? comparator-ordered?)
  (hash? comparator-hashable?))

```

comparator-test-type

索引

goldfish/srfi/srfi-128.scm

△ 4 ▽

```

;; Invoke the test type
(define (comparator-test-type comparator obj)
  ((comparator-type-test-predicate comparator) obj))

```

comparator-check-type

索引

goldfish/srfi/srfi-128.scm

[△](#) [5](#) ▾

```
(define (comparator-check-type comparator obj)
  (if (comparator-test-type comparator obj)
      #t
      (type-error "comparator type check failed" comparator obj)))
```

comparator-hash

索引

goldfish/srfi/srfi-128.scm

[△](#) [6](#) ▾

```
(define (comparator-hash comparator obj)
  ((comparator-hash-function comparator) obj))
```

12.3 测试

tests/goldfish/liii/comparator-test.scm

[1](#) ▾

```
(import (liii comparator)
        (liii check)
        (liii base))

(check-set-mode! 'report-failed)
```

12.4 内部函数

binary=?

goldfish/srfi/srfi-128.scm

[△](#) [7](#) ▾

```
(define (binary=? comparator a b)
  ((comparator-equality-predicate comparator) a b))
```

binary<?

goldfish/srfi/srfi-128.scm

[△](#) [8](#) ▾

```
(define (binary<? comparator a b)
  ((comparator-ordering-predicate comparator) a b))
```

binary>?

goldfish/srfi/srfi-128.scm

[△](#) [9](#) ▾

```
(define (binary>? comparator a b)
  (binary<? comparator b a))
```

binary<=?

goldfish/srfi/srfi-128.scm △ 10 ▾

```
(define (binary<=? comparator a b)
  (not (binary>? comparator a b)))
```

binary>=?

goldfish/srfi/srfi-128.scm △ 11 ▾

```
(define (binary>=? comparator a b)
  (not (binary<? comparator a b)))
```

%salt%

goldfish/srfi/srfi-128.scm △ 12 ▾

```
(define (%salt%)
  16064047)
```

hash-bound

$2^{25} - 1$

goldfish/srfi/srfi-128.scm △ 13 ▾

```
(define (hash-bound)
  33554432)
```

make-hashter

goldfish/srfi/srfi-128.scm △ 14 ▾

```
(define (make-hashter)
  (let ((result (%salt%)))
    (case-lambda
      (() result)
      ((n) (set! result (+ (modulo (* result 33) (hash-bound)) n)
                           result)))))
```

12.5 构造器

make-comparator 索引

goldfish/srfi/srfi-128.scm △ 15 ▾

```
(define (make-comparator type-test equality ordering hash)
  (make-raw-comparator
    (if (eq? type-test #t) (lambda (x) #t) type-test)
    (if (eq? equality #t) (lambda (x y) (eqv? (ordering x y) 0)) equality)
    (if ordering ordering (lambda (x y) (error "ordering not supported")))
    (if hash hash (lambda (x y) (error "hashing not supported")))
    (if ordering #t #f)
    (if hash #t #f)))
```

make-eq-comparator 索引

[goldfish/srfi/srfi-128.scm](#)
[△ 16 ▾](#)

```
(define (make-eq-comparator)
  (make-comparator #t eq? #f default-hash))
```

make-eqv-comparator

[goldfish/srfi/srfi-128.scm](#)
[△ 17 ▾](#)

```
(define (make-eqv-comparator)
  (make-comparator #t eqv? #f default-hash))
```

make-equal-comparator

[goldfish/srfi/srfi-128.scm](#)
[△ 18 ▾](#)

```
(define (make-equal-comparator)
  (make-comparator #t equal? #f default-hash))
```

make-pair-comparator

[goldfish/srfi/srfi-128.scm](#)
[△ 19 ▾](#)

```
(define (make-pair-type-test car-comparator cdr-comparator)
  (lambda (obj)
    (and (pair? obj)
         (comparator-test-type car-comparator (car obj))
         (comparator-test-type cdr-comparator (cdr obj))))))

(define (make-pair=? car-comparator cdr-comparator)
  (lambda (a b)
    (and ((comparator-equality-predicate car-comparator) (car a) (car b))
         ((comparator-equality-predicate cdr-comparator) (cdr a) (cdr b))))))

(define (make-pair<? car-comparator cdr-comparator)
  (lambda (a b)
    (if (=? car-comparator (car a) (car b))
        (<? cdr-comparator (cdr a) (cdr b))
        (<? car-comparator (car a) (car b))))))

(define (make-pair-hash car-comparator cdr-comparator)
  (lambda (obj)
    (let ((acc (make-hashtab)))
      (acc (comparator-hash car-comparator (car obj)))
      (acc (comparator-hash cdr-comparator (cdr obj)))
      (acc))))))

(define (make-pair-comparator car-comparator cdr-comparator)
  (make-comparator
    (make-pair-type-test car-comparator cdr-comparator)
    (make-pair=? car-comparator cdr-comparator)
    (make-pair<? car-comparator cdr-comparator)
    (make-pair-hash car-comparator cdr-comparator)))
```

make-list-comparator

```
;; Cheap test for listness
(define (norp? obj) (or (null? obj) (pair? obj)))

(define (make-list-comparator element-comparator type-test empty? head tail)
  (make-comparator
    (make-list-type-test element-comparator type-test empty? head tail)
    (make-list=? element-comparator type-test empty? head tail)
    (make-list<? element-comparator type-test empty? head tail)
    (make-list-hash element-comparator type-test empty? head tail)))

(define (make-list-type-test element-comparator type-test empty? head tail)
  (lambda (obj)
    (and
      (type-test obj)
      (let ((elem-type-test (comparator-type-test-predicate element-comparator)))
        (let loop ((obj obj))
          (cond
            ((empty? obj) #t)
            ((not (elem-type-test (head obj))) #f)
            (else (loop (tail obj))))))))))

(define (make-list=? element-comparator type-test empty? head tail)
  (lambda (a b)
    (let ((elem=? (comparator-equality-predicate element-comparator)))
      (let loop ((a a) (b b))
        (cond
          ((and (empty? a) (empty? b) #t))
          ((empty? a) #f)
          ((empty? b) #f)
          ((elem=? (head a) (head b)) (loop (tail a) (tail b)))
          (else #f))))))

(define (make-list<? element-comparator type-test empty? head tail)
  (lambda (a b)
    (let ((elem=? (comparator-equality-predicate element-comparator))
          (elem<? (comparator-ordering-predicate element-comparator)))
      (let loop ((a a) (b b))
        (cond
          ((and (empty? a) (empty? b) #f))
          ((empty? a) #t)
          ((empty? b) #f)
          ((elem=? (head a) (head b)) (loop (tail a) (tail b)))
          ((elem<? (head a) (head b)) #t)
          (else #f))))))

(define (make-list-hash element-comparator type-test empty? head tail)
  (lambda (obj)
    (let ((elem-hash (comparator-hash-function element-comparator))
          (acc (make-hashtab)))
      (let loop ((obj obj))
        (cond
          ((empty? obj) (acc))
          (else (acc (elem-hash (head obj)) (loop (tail obj)))))))))
```

make-vector-comparator[goldfish/srfi/srfi-128.scm](http://goldfish.srfi/srfi-128.scm)

△ 21 ▽

```

(define (make-vector-comparator element-comparator type-test length ref)
  (make-comparator
    (make-vector-type-test element-comparator type-test length ref)
    (make-vector=? element-comparator type-test length ref)
    (make-vector<? element-comparator type-test length ref)
    (make-vector-hash element-comparator type-test length ref)))

(define (make-vector-type-test element-comparator type-test length ref)
  (lambda (obj)
    (and
      (type-test obj)
      (let ((elem-type-test (comparator-type-test-predicate element-comparator))
            (len (length obj)))
        (let loop ((n 0))
          (cond
            ((= n len) #t)
            ((not (elem-type-test (ref obj n))) #f)
            (else (loop (+ n 1))))))))))

(define (make-vector=? element-comparator type-test length ref)
  (lambda (a b)
    (and
      (= (length a) (length b))
      (let ((elem=? (comparator-equality-predicate element-comparator))
            (len (length b)))
        (let loop ((n 0))
          (cond
            ((= n len) #t)
            ((elem=? (ref a n) (ref b n)) (loop (+ n 1)))
            (else #f)))))))

(define (make-vector<? element-comparator type-test length ref)
  (lambda (a b)
    (cond
      ((< (length a) (length b)) #t)
      ((> (length a) (length b)) #f)
      (else
        (let ((elem=? (comparator-equality-predicate element-comparator))
              (elem<? (comparator-ordering-predicate element-comparator))
              (len (length a)))
          (let loop ((n 0))
            (cond
              ((= n len) #f)
              ((elem=? (ref a n) (ref b n)) (loop (+ n 1)))
              ((elem<? (ref a n) (ref b n)) #t)
              (else #f))))))))))

(define (make-vector-hash element-comparator type-test length ref)
  (lambda (obj)
    (let ((elem-hash (comparator-hash-function element-comparator))
          (acc (make-hashtab))
          (len (length obj)))
      (let loop ((n 0))
        (cond
          ((= n len) (acc))
          (else (acc (elem-hash (ref obj n)) (loop (+ n 1))))))))))

```

12.6 标准函数

object-type

goldfish/srfi/srfi-128.scm

△ 22 ▽

```
(define (object-type obj)
  (cond
    ((null? obj) 0)
    ((pair? obj) 1)
    ((boolean? obj) 2)
    ((char? obj) 3)
    ((string? obj) 4)
    ((symbol? obj) 5)
    ((number? obj) 6)
    ((vector? obj) 7)
    ((bytevector? obj) 8)
    (else 65535)))
```

boolean<?



goldfish/srfi/srfi-128.scm

△ 23 ▽

```
(define (boolean<? a b)
  ;; #f < #t but not otherwise
  (and (not a) b))
```

complex<?

goldfish/srfi/srfi-128.scm

△ 24 ▽

```
(define (complex<? a b)
  (if (= (real-part a) (real-part b))
      (< (imag-part a) (imag-part b))
      (< (real-part a) (real-part b))))
```

symbol<?

goldfish/srfi/srfi-128.scm

△ 25 ▽

```
(define (symbol<? a b)
  (string<? (symbol->string a) (symbol->string b)))
```

boolean-hash



char-hash



char-ci-hash



string-hash



string-ci-hash



symbol-hash



number-hash

索引

default-hash

索引

goldfish/srfi/srfi-128.scm

△ 26 ▽

```
(define boolean-hash hash-code)
(define char-hash hash-code)
(define char-ci-hash hash-code)
(define string-hash hash-code)
(define string-ci-hash hash-code)
(define symbol-hash hash-code)
(define number-hash hash-code)
(define default-hash hash-code)
```

default-ordering

goldfish/srfi/srfi-128.scm

△ 27 ▽

```
(define (dispatch-ordering type a b)
  (case type
    ((0) 0) ; All empty lists are equal
    ((1) ((make-pair=? (make-default-comparator) (make-default-comparator)) a b))
    ((2) (boolean=? a b))
    ((3) (char=? a b))
    ((4) (string=? a b))
    ((5) (symbol=? a b))
    ((6) (complex=? a b))
    ((7) ((make-vector=? (make-default-comparator) vector? vector-length vector-ref)
          a b))
    ((8) ((make-vector=? (make-comparator exact-integer? = < default-hash)
                          bytevector? bytevector-length bytevector-u8-ref)
          a b))
    ; Add more here
    (else (binary=? (registered-comparator type) a b))))

(define (default-ordering a b)
  (let ((a-type (object-type a))
        (b-type (object-type b)))
    (cond
      ((< a-type b-type) #t)
      ((> a-type b-type) #f)
      (else (dispatch-ordering a-type a b)))))
```

default-equality

goldfish/srfi/srfi-128.scm

△ 28 ▾

```

(define (dispatch-equality type a b)
  (case type
    ((0) #t) ; All empty lists are equal
    ((1) ((make-pair=? (make-default-comparator) (make-default-comparator)) a b))
    ((2) (boolean=? a b))
    ((3) (char=? a b))
    ((4) (string=? a b))
    ((5) (symbol=? a b))
    ((6) (= a b))
    ((7) ((make-vector=? (make-default-comparator)
                          vector? vector-length vector-ref) a b))
    ((8) ((make-vector=? (make-comparator exact-integer? = < default-hash)
                          bytevector? bytevector-length bytevector-u8-ref) a b))
    ; Add more here
    (else (binary=? (registered-comparator type) a b))))

(define (default-equality a b)
  (let ((a-type (object-type a))
        (b-type (object-type b)))
    (if (= a-type b-type)
        (dispatch-equality a-type a b)
        #f)))

```

make-default-comparator

索引

goldfish/srfi/srfi-128.scm

△ 29 ▾

```

(define (make-default-comparator)
  (make-comparator
   (lambda (obj) #t)
   default-equality
   default-ordering
   default-hash))

```

tests/goldfish/liii/comparator-test.scm

△ 2 ▾

```

(let1 default-comp (make-default-comparator)
  (check-false (<? default-comp #t #t))
  (check-false (<? default-comp #f #f))
  (check-true (<? default-comp #f #t))
  (check-false (<? default-comp #t #f))
  (check-true (<? default-comp (cons #f #f) (cons #t #t)))
  (check-true (<? default-comp (list 1 2) (list 2 3)))
  (check-true (<? default-comp (list 1 2) (list 1 3)))
  (check-true (<? default-comp (vector "a" "b") (vector "b" "c")))

  (check-false (<? default-comp 1 1))
  (check-true (<? default-comp 0+1i 0+2i))
  (check-true (<? default-comp 1+2i 2+2i))

  (check (comparator-hash default-comp (list 1 2)) => 42)
)

```

12.7 比较谓词

=?

索引

goldfish/srfi/srfi-128.scm

△ 30 ▽

```
(define (=? comparator a b . objs)
  (let loop ((a a) (b b) (objs objs))
    (and (binary=? comparator a b)
         (if (null? objs) #t (loop b (car objs) (cdr objs))))))
```

<?

索引

goldfish/srfi/srfi-128.scm

△ 31 ▽

```
(define (<? comparator a b . objs)
  (let loop ((a a) (b b) (objs objs))
    (and (binary<? comparator a b)
         (if (null? objs) #t (loop b (car objs) (cdr objs))))))
```

>?

索引

goldfish/srfi/srfi-128.scm

△ 32 ▽

```
(define (>? comparator a b . objs)
  (let loop ((a a) (b b) (objs objs))
    (and (binary>? comparator a b)
         (if (null? objs) #t (loop b (car objs) (cdr objs))))))
```

<=?

索引

goldfish/srfi/srfi-128.scm

△ 33 ▽

```
(define (<=? comparator a b . objs)
  (let loop ((a a) (b b) (objs objs))
    (and (binary<=? comparator a b)
         (if (null? objs) #t (loop b (car objs) (cdr objs))))))
```

>=?

索引

goldfish/srfi/srfi-128.scm

△ 34 ▽

```
(define (>=? comparator a b . objs)
  (let loop ((a a) (b b) (objs objs))
    (and (binary>=? comparator a b)
         (if (null? objs) #t (loop b (car objs) (cdr objs))))))
```

12.8 结尾

goldfish/liii/comparator.scm

△ 3

```
) ; end of begin
) ; end of define-library
```

goldfish/srfi/srfi-128.scm

△ 35

```
) ; end of begin
) ; end of define-library
```

tests/goldfish/liii/comparator-test.scm

△ 3

```
(check-report)
```


第 13 章

(liii hash-table)

13.1 许可证

goldfish/srfi/srfi-125.scm1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

goldfish/liii/hash-table.scm1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

tests/goldfish/liii/hash-table-test.scm

1 ▾

```

;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;

```

13.2 测试

tests/goldfish/liii/hash-table-test.scm

△ 2 ▾

```

(import (liii check)
        (liii comparator)
        (liii hash-table)
        (liii base))

(check-set-mode! 'report-failed)

(define empty-ht (make-hash-table))

```

13.3 接口

goldfish/srfi/srfi-125.scm

△ 2 ▾

```

(define-library (srfi srfi-125)
  (import (srfi srfi-1)
          (liii base)
          (liii error))
  (export
    make-hash-table hash-table hash-table-unfold alist->hash-table
    hash-table? hash-table-contains? hash-table-empty? hash-table=?
    hash-table-mutable?
    hash-table-ref hash-table-ref/default
    hash-table-set! hash-table-delete! hash-table-intern! hash-table-update!
    hash-table-update!/default hash-table-pop! hash-table-clear!
    hash-table-size hash-table-keys hash-table-values hash-table-entries
    hash-table-find hash-table-count
    hash-table-for-each hash-table-map->list
    hash-table->alist
  )
  (begin

```

goldfish/liii/hash-table.scm

△ 2 ▽

```
(define-library (liii hash-table)
  (import (srfi srfi-125)
          (srfi srfi-128))
  (export
    make-hash-table hash-table hash-table-unfold alist->hash-table
    hash-table? hash-table-contains? hash-table-empty? hash-table=?
    hash-table-mutable?
    hash-table-ref hash-table-ref/default
    hash-table-set! hash-table-delete! hash-table-intern! hash-table-update!
    hash-table-update!/default hash-table-pop! hash-table-clear!
    hash-table-size hash-table-keys hash-table-values hash-table-entries
    hash-table-find hash-table-count
    hash-table-for-each hash-table-map->list
    hash-table->alist
  )
  (begin
```

13.3.1 访问哈希表中的元素

除了SRFI 125定义的`hash-table-ref`和`hash-table-ref/default`之外，我们可以用S7 Scheme内置的访问方式：

tests/goldfish/liii/hash-table-test.scm

△ 3 ▽

```
(let1 ht (make-hash-table)
  (check (ht 'a) => #f)
  (hash-table-set! ht 'a 1)
  (check (ht 'a) => 1))
```

13.4 实现

13.4.1 子函数

goldfish/srifi/srifi-125.scm

△ 3 ▽

```
(define (assert-hash-table-type ht f)
  (when (not (hash-table? ht))
    (error 'type-error f "this parameter must be typed as hash-table")))

(define s7-hash-table-set! hash-table-set!)
(define s7-make-hash-table make-hash-table)
(define s7-hash-table-entries hash-table-entries)
```

13.4.2 构造器

make-hash-table

索引

goldfish/srifi/srifi-125.scm

△ 4 ▽

```
(define (make-hash-table . args)
  (cond ((null? args) (s7-make-hash-table))
        ((comparator? (car args))
         (let* ((equiv (comparator-equality-predicate (car args)))
                (hash-func (comparator-hash-function (car args))))
           (s7-make-hash-table 8 (cons equiv hash-func) (cons #t #t))))
        (else (type-error "make-hash-table"))))
```

tests/goldfish/liii/hash-table-test.scm

△ 4 ▽

```
(let1 ht (make-hash-table (make-default-comparator))
  (hash-table-set! ht 1 2)
  (check (ht 1) => 2))

(let* ((mod10 (lambda (x) (modulo x 10)))
      (digit=? (lambda (x y) (= (modulo x 10) (modulo y 10))))
      (comp (make-comparator number? digit=? #f mod10))
      (ht (make-hash-table comp)))
  (hash-table-set! ht 1 2)
  (hash-table-set! ht 11 3)
  (check (ht 1) => 3)
  (check (ht 11) => 3)
  (check (ht 21) => 3))
```

hash-table

alist->hash-table

索引

goldfish/srfi/srfi-125.scm

△ 5 ▽

```
(define alist->hash-table
  (typed-lambda ((lst list?))
    (when (odd? (length lst))
      (value-error "The length of lst must be even!"))
    (let1 ht (make-hash-table)
      (let loop ((rest lst))
        (if (null? rest)
            ht
            (begin
              (hash-table-set! ht (car rest) (cadr rest))
              (loop (cddr rest)))))))
```

13.4.3 谓词

hash-table?

索引

S7内置函数。判断一个对象是不是哈希表。

hash-table-contains? ((ht hash-table?) key) => bool

索引

实现

goldfish/srfi/srfi-125.scm

△ 6 ▽

```
(define (hash-table-contains? ht key)
  (not (not (hash-table-ref ht key))))
```

测试

tests/goldfish/liii/hash-table-test.scm

△ 5 ▽

```
(let1 ht (make-hash-table)
  (hash-table-set! ht 'brand 'liii)
  (check (hash-table-contains? ht 'brand) => #t)
  (hash-table-set! ht 'brand #f)
  (check (hash-table-contains? ht 'brand) => #f))
```

hash-table-empty?

索引

实现

goldfish/srfi/srfi-125.scm

△ 7 ▽

```
(define (hash-table-empty? ht)
  (zero? (hash-table-size ht)))
```


测试

tests/goldfish/liii/hash-table-test.scm

△ 6 ▽

```
(check (hash-table-empty? empty-ht) => #t)

(let1 test-ht (make-hash-table)
  (hash-table-set! test-ht 'key 'value)
  (check (hash-table-empty? test-ht) => #f))
```

hash-table=?  (ht1 ht2) => boolean

实现

goldfish/srfi/srfi-125.scm

△ 8 ▽

```
(define (hash-table=? ht1 ht2)
  (equal? ht1 ht2))
```

测试

tests/goldfish/liii/hash-table-test.scm

△ 7 ▽

```
(let ((empty-h1 (make-hash-table))
      (empty-h2 (make-hash-table)))
  (check (hash-table=? empty-h1 empty-h2) => #t))

(let ((t1 (make-hash-table))
      (t2 (make-hash-table)))
  (hash-table-set! t1 'a 1)
  (hash-table-set! t2 'a 1)
  (check (hash-table=? t1 t2) => #t)
  (hash-table-set! t1 'b 2)
  (check (hash-table=? t1 t2) => #f))
```

13.4.4 选择器

hash-table-ref  (hash-table-ref ht key) => value

ht. 哈希表

key. 键

value. 返回hash表中key这个键对应的值

SRFI 125定义的hash-table-ref的函数签名是这样的：(hash-table-ref hash-table key [failure [success]])。两参数形式的hash-table-ref是S7 Scheme的内置函数。

在S7 Scheme中，可以直接将hash-table视作一个单参数的函数，比如(ht 'key)等价于(hash-table-ref ht 'key)。

tests/goldfish/liii/hash-table-test.scm

△ 8 ▽

```
(check (hash-table-ref empty-ht 'key) => #f)

(let1 ht (make-hash-table)
  (hash-table-set! ht 'key 'value)
  (check (hash-table-ref ht 'key) => 'value)
  (check (ht 'key) => 'value))
```

索引

hash-table-ref/default (`(hash-table-ref/default ht key default)`) => value

ht. 哈希表

key. 键

default. 默认值，如果key这个键在哈希表中对应的值不存在，则返回默认值。注意，该默认值只有在key这个键不存在的时候，才会被求值。

value. 键对应的值，如果不存在，则为默认值。

测试

当键对应的值存在时，default不会被求值，故而测试中的(`display "hello"`)实际不会被执行。

tests/goldfish/liii/hash-table-test.scm

△ 9 ▽

```
(let1 ht (make-hash-table)
  (check (hash-table-ref/default ht 'key 'value1) => 'value1)
  (check (hash-table-ref/default ht 'key (+ 1 2)) => 3)

  (hash-table-set! ht 'key 'value)
  (check (hash-table-ref/default ht 'key
    (begin (display "hello")
      (+ 1 2)))
    => 'value)
) ; end of let1
```

实现

goldfish/srfi/srfi-125.scm

△ 9 ▽

```
(define-macro (hash-table-ref/default ht key default)
  `(or (hash-table-ref ,ht ,key)
    ,default))
```

13.4.5 修改器

索引

hash-table-set!

测试

tests/goldfish/liii/hash-table-test.scm

△ 10 ▽

```
(let1 ht (make-hash-table)
  (hash-table-set! ht 'k1 'v1 'k2 'v2)
  (check (ht 'k1) => 'v1)
  (check (ht 'k2) => 'v2)
)
```

实现

goldfish/srfi/srfi-125.scm

△ 10 ▽

```
(define (hash-table-set! ht . rest)
  (assert-hash-table-type ht hash-table-set!)
  (let1 len (length rest)
    (when (or (odd? len) (zero? len))
      (error 'wrong-number-of-args len "but must be even and non-zero")))

  (s7-hash-table-set! ht (car rest) (cadr rest))
  (when (> len 2)
    (apply hash-table-set! (cons ht (cddr rest))))))
```

hash-table-delete!



测试

tests/goldfish/liii/hash-table-test.scm

△ 11 ▾

```
(let1 ht (make-hash-table)
  (hash-table-update! ht 'key 'value)
  (check (hash-table-delete! ht 'key) => 1)
  (check-false (hash-table-contains? ht 'key))

  (hash-table-update! ht 'key1 'value1)
  (hash-table-update! ht 'key2 'value2)
  (hash-table-update! ht 'key3 'value3)
  (hash-table-update! ht 'key4 'value4)
  (check (hash-table-delete! ht 'key1 'key2 'key3) => 3)
)
```

实现

goldfish/srfi/srfi-125.scm

△ 11 ▾

```
(define (hash-table-delete! ht key . keys)
  (assert-hash-table-type ht hash-table-delete!)
  (let1 all-keys (cons key keys)
    (length
     (filter
      (lambda (x)
        (if (hash-table-contains? ht x)
            (begin
              (s7-hash-table-set! ht x #f)
              #t)
            #f)))
      all-keys))))
```

hash-table-update!



实现

goldfish/srfi/srfi-125.scm

△ 12 ▾

```
(define (hash-table-update! ht key value)
  (hash-table-set! ht key value))
```

测试

tests/goldfish/liii/hash-table-test.scm

△ 12 ▾

```
(let1 ht (make-hash-table)
  (hash-table-update! ht 'key 'value)
  (check (ht 'key) => 'value)
  (hash-table-update! ht 'key 'value1)
  (check (ht 'key) => 'value1)
  (hash-table-update! ht 'key #f)
  (check (ht 'key) => #f))
```

hash-table-clear!



实现

goldfish/srfi/srfi-125.scm

△ 13 ▾

```
(define (hash-table-clear! ht)
  (for-each
    (lambda (key)
      (hash-table-set! ht key #f))
    (hash-table-keys ht)))
```

测试

tests/goldfish/liii/hash-table-test.scm

△ 13 ▾

```
(let1 ht (make-hash-table)
  (hash-table-update! ht 'key 'value)
  (hash-table-update! ht 'key1 'value1)
  (hash-table-update! ht 'key2 'value2)
  (hash-table-clear! ht)
  (check-true (hash-table-empty? ht)))
```

13.4.6 哈希表整体

hash-table-size

索引

实现

goldfish/srfi/srfi-125.scm

△ 14 ▾

```
(define hash-table-size s7-hash-table-entries)
```

测试

tests/goldfish/liii/hash-table-test.scm

△ 14 ▾

```
(check (hash-table-size empty-ht) => 0)

(let1 populated-ht (make-hash-table)
  (hash-table-set! populated-ht 'key1 'value1)
  (hash-table-set! populated-ht 'key2 'value2)
  (hash-table-set! populated-ht 'key3 'value3)
  (check (hash-table-size populated-ht) => 3))
```

hash-table-keys

索引

实现

goldfish/srfi/srfi-125.scm

△ 15 ▾

```
(define (hash-table-keys ht)
  (map car ht))
```

测试

tests/goldfish/liii/hash-table-test.scm

△ 15 ▾

```
(check (hash-table-keys empty-ht) => '())

(let1 ht (make-hash-table)
  (hash-table-set! ht 'k1 'v1)
  (check (hash-table-keys ht) => '(k1)))
```

hash-table-values

索引

实现

goldfish/srfi/srfi-125.scm

△ 16 ▾

```
(define (hash-table-values ht)
  (map cdr ht))
```

测试

tests/goldfish/liii/hash-table-test.scm

△ 16 ▾

```
(check (hash-table-values empty-ht) => '())

(let1 ht (make-hash-table)
  (hash-table-set! ht 'k1 'v1)
  (check (hash-table-values ht) => '(v1)))
```

hash-table-entries

索引

goldfish/srfi/srfi-125.scm

△ 17 ▾

```
(define hash-table-entries
  (typed-lambda ((ht hash-table?))
    (let ((ks (hash-table-keys ht))
          (vs (hash-table-values ht)))
      (values ks vs))))
```

tests/goldfish/liii/hash-table-test.scm

△ 17 ▾

```
(let1 ht (make-hash-table)
  (check (call-with-values (lambda () (hash-table-entries ht))
                           (lambda (ks vs) (list ks vs)))
    => (list (list ) (list )))

  (hash-table-set! ht 'k1 'v1)
  (check (call-with-values (lambda () (hash-table-entries ht))
                           (lambda (ks vs) (list ks vs)))
    => (list (list 'k1) (list 'v1))))
```

hash-table-find

索引

(proc (ht hash-table?) failure) => obj

对于哈希表中的每个关联项，调用proc函数处理它的键和值。如果proc返回真（true），那么hash-table-find函数将返回proc返回的结果。如果所有对proc的调用都返回假（#f），则返回调用thunk failure函数的结果。

goldfish/srfi/srfi-125.scm

△ 18 ▾

tests/goldfish/liii/hash-table-test.scm

△ 18 ▾

hash-table-count

索引

实现

goldfish/srfi/srfi-125.scm

△ 19 ▾

```
(define hash-table-count
  (typed-lambda ((pred? procedure?) (ht hash-table?))
    (count (lambda (x) (pred? (car x) (cdr x)))
      (map values ht))))
```

测试

tests/goldfish/liii/hash-table-test.scm

△ 19 ▽

```

(check (hash-table-count (lambda (k v) #f) (hash-table))) => 0)
(check (hash-table-count (lambda (k v) #t) (hash-table 'a 1 'b 2 'c 3))) => 3)
(check (hash-table-count (lambda (k v) #f) (hash-table 'a 1 'b 2 'c 3))) => 0)

(check (hash-table-count (lambda (k v) (eq? k 'b)) (hash-table 'a 1 'b 2 'c 3))) => 1)

(check (hash-table-count (lambda (k v) (> v 1)) (hash-table 'a 1 'b 2 'c 3))) => 2)

(check (hash-table-count (lambda (k v) (string? k))
                          (hash-table "apple" 1 "banana" 2))) => 2)

(check (hash-table-count (lambda (k v) (and (symbol? k) (even? v)))
                          (hash-table 'apple 2 'banana 3 'cherry 4))) => 2)

(check (hash-table-count (lambda (k v) (eq? k v))
                          (hash-table 'a 'a 'b 'b 'c 'd))) => 2)

(check (hash-table-count (lambda (k v) (number? k))
                          (hash-table 1 100 2 200 3 300))) => 3)

(check (hash-table-count (lambda (k v) (list? v))
                          (hash-table 'a '(1 2) 'b '(3 4) 'c 3))) => 2)

(check (hash-table-count (lambda (k v)
                              (= (char->integer (string-ref (symbol->string k) 0)) v))
                          (hash-table 'a 97 'b 98 'c 99))) => 3)

```

13.4.7 映射和折叠

索引

```
hash-table-foreach ((proc procedure?) (ht hash-table?)) => #<unspecified>
```

对哈希表中的每个关联调用 `proc`，传递两个参数：关联的键和关联的值。`proc` 返回的值被丢弃。返回一个未指定的值。

goldfish/srfi/srfi-125.scm

△ 20 ▽

```

(define hash-table-for-each
  (typed-lambda ((proc procedure?) (ht hash-table?))
    (for-each (lambda (x) (proc (car x) (cdr x)))
              ht)))

```

tests/goldfish/liii/hash-table-test.scm

△ 20 ▽

```

(let1 cnt 0
  (hash-table-for-each
   (lambda (k v)
     (set! cnt (+ cnt v)))
   (hash-table 'a 1 'b 2 'c 3))
  (check cnt => 6))

```

索引

```
hash-table-map->list ((proc procedure?) (ht hash-table?)) => list
```

对哈希表中的每个关联调用 `proc`，传递两个参数：关联的键和关联的值。每次调用 `proc` 返回的值会被累积到一个列表中，并最终返回该列表。

goldfish/srfi/srfi-125.scm

△ 21 ▾

```
(define hash-table-map->list
  (typed-lambda ((proc procedure?) (ht hash-table?))
    (map (lambda (x) (proc (car x) (cdr x)))
         ht)))
```

tests/goldfish/liii/hash-table-test.scm

△ 21 ▾

```
(let* ((ht (hash-table 'a 1 'b 2 'c 3))
      (ks (hash-table-map->list (lambda (k v) k) ht))
      (vs (hash-table-map->list (lambda (k v) v) ht)))
  (check-true (in? 'a ks))
  (check-true (in? 'b ks))
  (check-true (in? 'c ks))
  (check-true (in? 1 vs))
  (check-true (in? 2 vs))
  (check-true (in? 3 vs)))
```

13.4.8 复制和转换

hash-table->alist

索引

goldfish/srfi/srfi-125.scm

△ 22 ▾

```
(define hash-table->alist
  (typed-lambda ((ht hash-table?))
    (append-map
     (lambda (x) (list (car x) (cdr x)))
     (map values ht))))
```

tests/goldfish/liii/hash-table-test.scm

△ 22 ▾

```
(let1 ht (make-hash-table)
  (check (hash-table->alist ht) => (list))
  (hash-table-set! ht 'k1 'v1)
  (check (hash-table->alist ht) => '(k1 v1)))

(check (hash-table->alist (alist->hash-table (list 'k1 'v1)))
      => (list 'k1 'v1))
```

13.5 结尾

tests/goldfish/liii/hash-table-test.scm

△ 23

```
(check-report)
```

goldfish/srfi/srfi-125.scm

△ 23

```
) ; end of begin
) ; end of define-library
```

goldfish/liii/hash-table.scm

△ 3

```
) ; end of begin
) ; end of library
```


第 14 章

(liii set)

14.1 许可证

[goldfish/liii/set.scm](#)

1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

[goldfish/srfi/srfi-113.scm](#)

1 ▾

```
; Copyright (C) John Cowan (2015). All Rights Reserved.
;
; Permission is hereby granted, free of charge, to any person obtaining a copy of
; this software and associated documentation files (the "Software"), to deal in
; the Software without restriction, including without limitation the rights to
; use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
; of the Software, and to permit persons to whom the Software is furnished to do
; so, subject to the following conditions:
;
; The above copyright notice and this permission notice shall be included in all
; copies or substantial portions of the Software.
;
; THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
; IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
; FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
; AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
; LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
; OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
; SOFTWARE.
;
```

14.2 接口

goldfish/liii/set.scm

△ 2 ▽

```
(define-library (liii set)
  (import (srfi srfi-113))
  (export set?)
  (begin
```

goldfish/srfi/srfi-113.scm

△ 2 ▽

```
(define-library (srfi srfi-113)
  (import (scheme base))
  (export set?)
  (begin
```

14.3 结尾

goldfish/liii/set.scm

△ 3

```
) ; end of begin
) ; end of define-library
```

goldfish/srfi/srfi-113.scm

△ 3

```
) ; end of begin
) ; end of define-library
```

第 15 章

三鲤扩展库说明

三鲤扩展库都是形如 (l_{iii} xyz) 的 Scheme 库。

15.1 结尾

[goldfish/scheme/base.scm](#)

△ 36

```
) ; end of begin  
) ; end of define-library
```

第 16 章

(scheme case-lambda)

16.1 协议

tests/goldfish/scheme/case-lambda-test.scm

1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

16.2 接口

在 Scheme 语言中，`case-lambda` 是一种特殊的 `lambda` 表达式，它可以根据不同数量的参数执行不同的代码块。`case-lambda` 允许你定义一个函数，这个函数根据传入参数的数量来选择执行不同的 `lambda` 表达式。

下面是一个使用 `case-lambda` 的示例：

tests/goldfish/scheme/case-lambda-test.scm

△ 2

```
(import (liii list)
        (liii check)
        (scheme case-lambda))

(check-set-mode! 'report-failed)

(define (my-func . args)
  (case-lambda
    (() "zero args")
    ((x) (+ x x))
    ((x y) (+ x y))
    ((x y . rest) (reduce + 0 (cons x (cons y rest))))))

(check (my-func) => "zero args")
(check (my-func) 2) => 4)
(check (my-func) 3 4) => 7)
(check (my-func) 1 2 3 4) => 10)

(check-report)
```

16.3 实现

goldfish/scheme/case-lambda.scm

```
; 0-clause BSD
; Bill Schottstaedt
; from S7 source repo: r7rs.scm

(define-library (scheme case-lambda)
  (export case-lambda)
  (begin

;; case-lambda
(define-macro (case-lambda . choices)
  `(lambda args
     (case (length args)
       ,@(map (lambda (choice)
                 (if (or (symbol? (car choice))
                        (negative? (length (car choice))))
                     ' (else (apply (lambda ,(car choice) ,(cdr choice)) args))
                     ' ((, (length (car choice))
                        (apply (lambda ,(car choice) ,(cdr choice)) args))))
              choices))))

) ; end of begin
) ; end of define-library
```

第 17 章

(scheme char)

17.1 许可证

tests/goldfish/scheme/char-test.scm1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

17.2 接口

tests/goldfish/scheme/char-test.scm△ 2 ▾

```
(import (liii check))

(check-set-mode! 'report-failed)
```

17.3 实现

S7ref

char-upcase

索引

S7内置函数。

tests/goldfish/scheme/char-test.scm

△ 3 ▽

```
(check (char-upcase #\A) => #\A)
(check (char-upcase #\a) => #\A)
(check (char-upcase #\?) => #\?)
(check (char-upcase #\$) => #\$)
(check (char-upcase #\.) => #\.)
(check (char-upcase #\\) => #\\)
(check (char-upcase #\5) => #\5)
(check (char-upcase #\) => #\))
(check (char-upcase #\%) => #\%)
(check (char-upcase #\0) => #\0)
(check (char-upcase #\_ ) => #\_ )
(check (char-upcase #\?) => #\?)
(check (char-upcase #\space) => #\space)
(check (char-upcase #\newline) => #\newline)
(check (char-upcase #\null) => #\null)
```

17.4 结尾

tests/goldfish/scheme/char-test.scm

△ 4

(check-report)

第 18 章

(scheme file)

18.1 许可证

goldfish/scheme/file.scm1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

18.2 接口

goldfish/scheme/file.scm△ 2 ▾

```
(define-library (scheme file)
  (export open-binary-input-file open-binary-output-file)
  (begin
```

18.3 实现

x7rs 索引

open-input-file

S7内置函数。

x7rs 索引

open-binary-input-file

goldfish/scheme/file.scm△ 3 ▾

```
(define open-binary-input-file open-input-file)
```

x7rs 索引

open-output-file

S7内置函数。

r7rs

索引

open-binary-output-file

goldfish/scheme/file.scm

△ 4 ▾

(define open-binary-output-file open-output-file)

18.4 结尾

goldfish/scheme/file.scm

△ 5

) ; end of begin
) ; end of define-library

第 19 章

(srfi sicp)

19.1 许可证

goldfish/srfi/srfi-216.scm1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

goldfish/srfi/sicp.scm1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

[tests/goldfish/srfi/sicp-test.scm](#)

1 ▾

```
;
; Copyright (C) 2024 The Goldfish Scheme Authors
;
; Licensed under the Apache License, Version 2.0 (the "License");
; you may not use this file except in compliance with the License.
; You may obtain a copy of the License at
;
; http://www.apache.org/licenses/LICENSE-2.0
;
; Unless required by applicable law or agreed to in writing, software
; distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
; WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
; License for the specific language governing permissions and limitations
; under the License.
;
```

19.2 接口

[goldfish/srfi/srfi-216.scm](#)

△ 2 ▾

```
(define-library (srfi srfi-216)
  (export true false nil runtime)
  (import (scheme time))
  (begin
```

[goldfish/srfi/sicp.scm](#)

△ 2

```
(define-library (srfi sicp)
  (export true false nil runtime)
  (import (srfi srfi-216)))
```

19.3 测试

[tests/goldfish/srfi/sicp-test.scm](#)

△ 2

```
(import (srfi sicp)
  (liii os)
  (liiii check))

(display (runtime))
(newline)

(when (os-linux?)
  (os-call "sleep 0.01"))

(display (runtime))
(newline)

(check-true true)
(check-false false)
(check-true (null? nil))

(check-report)
```

19.4 实现

goldfish/srfi/srfi-216.scm

△ 3 ▽

```
(define true #t)

(define false #f)

(define nil '())

(define (runtime)
  (round (* 1000 (current-second))))
```

19.5 结尾

goldfish/srfi/srfi-216.scm

△ 4

```
) ; end of begin
) ; end of define-library
```

索引

!=	34	delete-duplicates	102
==	33	delete-file	11
=?	144	display*	34
???	40	drop	90
alist->hash-table	148	drop-right	90
and	18	drop-while	100
and-let*	20	eighth	89
any	101	eof-object	33
append	92	every	101
append-map	96	exact?	23
apply	29	fifth	88
binary-port?	32	file-error?	31
boolean=?	27	file-exists?	11
boolean-hash	141	file-exists-error	39
boolean<?	141	file-not-found-error	38
caar	87	filter	97
call-with-port	31	find	99
car	86	first	87
case	18	flatmap	106
case*	51	floor	23
cdr	86	floor-quotient	25
char?	28	fold	93
char=?	28	fold-right	94
char-ci-hash	141	for-each	93
char-hash	141	fourth	88
char-upcase	163	gcd	26
check	47	>?	144
check-catch	47	>=?	144
check-failed?	45	guard	30
check-false	47	hash-table?	148
check-report	48	hash-table=?	149
check-set-mode!	43	hash-table-clear!	151
check-true	47	hash-table-contains?	148
circular-list	84	hash-table-count	153
circular-list?	85	hash-table-delete!	151
close-input-port	32	hash-table-empty?	148
close-output-port	32	hash-table-entries	153
close-port	32	hash-table-find	153
comparator?	135	hash-table-foreach	154
comparator-check-type	135	hash-table->alist	155
comparator-equality-predicate	135	hash-table-keys	152
comparator-hash	136	hash-table-map->list	154
comparator-hashable?	135	hash-table-ref	149
comparator-hash-function	135	hash-table-ref/default	150
comparator-ordered?	135	hash-table-set!	150
comparator-ordering-predicate	135	hash-table-size	152
comparator-test-type	135	hash-table-update!	151
comparator-type-test-predicate	135	hash-table-values	152
compose	35	identity	35
cons	83	import	12
count	92	in?	34
default-hash	142	input-port?	32
define-library	12	input-port-open?	32
define-record-type	21	int-vector	111
delete	102	iota	84

last	91	reduce	95
last-pair	91	reduce-right	96
length	92	remove	97
length=?	103	second	88
length>?	103	seventh	88
length>=?	104	sixth	88
<?	144	square	27
<=?	144	stack	126
let	19	stack-empty?	127
let1	35	stack->list	127
letrec*	19	stack-pop!	127
let-values	20	stack-push!	127
list	83	stack-size	127
list?	85	stack-top	127
list->string	56	string?	56
list->vector	121	string-any	61
list-index	100	string-append	76
list-not-null?	106	string-ci-hash	141
list-ref	87	string-contains	75
list-view	104	string-copy	63
make-comparator	137	string-count	75
make-default-comparator	143	string-drop	65
make-eq-comparator	137	string-drop-right	65
make-equal-comparator	138	string-every	58
make-eqv-comparator	138	string-for-each	78
make-hash-table	147	string->list	56
make-list	83	string->vector	122
make-list-comparator	138	string-hash	141
make-pair-comparator	138	string-index	74
make-vector	111	string-index-right	74
make-vector-comparator	140	string-join	57
map	93	string-length	63
member	98	string-map	77
memq	98	string-null?	58
memv	98	string-pad	66
ninth	89	string-pad-right	67
not-a-directory-error	38	string-prefix?	73
not-null-list?	106	string-ref	63
null?	85	string-reverse	76
null-list?	86, 106	string-suffix?	73
number?	23	string-take	65
number-hash	142	string-take-right	65
open-binary-input-file	165	string-tokenize	78
open-binary-output-file	166	string-trim	69
open-input-file	165	string-trim-both	73
open-output-file	165	string-trim-right	70
or	19	symbol-hash	141
os-error	38	take	89
output-port?	32	take-right	90
output-port-open?	32	take-while	99
pair?	84	tenth	89
partition	97	test	48
port?	32	textual-port?	32
procedure?	29	third	88
queue	131	timeout-error	39
queue-back	131	typed-lambda	35
queue-empty?	131	type-error	39
queue-front	131	type-error?	39
queue->list	132	value-error	39
queue-pop!	132	vector	111
queue-push!	132	vector?	113
queue-size	131	vector-any	115
quotient	26	vector-append	112
read	32	vector-copy	112
read-error?	31	vector-copy!	118
receive	29	vector-count	115

vector-empty?	113	vector-index-right	116
vector-every	116	vector-length	114
vector-fill!	118	vector-map	114
vector-for-each	114	vector-partition	117
vector->list	121	vector-ref	114
vector->string	121	vector-set!	117
vector-index	116	vector-swap!	118