Scheme Cheat Sheet

Basic types

- Booleans: `#t`, `#f`
- Numbers: `132897132989731289713`, `1.23e+33`, `23/169`, `12+4i`
- Characters: `#\a`, `#\Z`
- Symbols: `a-symbol`, `another-symbol?`, `indeed!`
- Vectors: `#(1 2 3 4)`
- Strings: `"this is a string"`
- Pairs and Lists: `(1 2 #\a)`, `(a . b)`, `()`

Lambdas

```
(lambda (x y)
    (+ (* x x ) (* y y )))
```

Example usage

```
((lambda (x y) (+ (* x x) (* y y))) 2 3) ; => 13
```

Bindings: let, let*, and letrec

These three binding forms create local variables. They share the same syntax, here explained with let:

```
(let ((<var> <expr>) ...) <body>)
```

let: the "parallel" let

In the case of the `let` form, the scope of the variables is limited to the expression `<body>`. Variables are bound "in parallel", e.g.:

```
(let ((x 1) (y 2))
    (let ((x y) (y x)) ; swap x and y
        x)) ; => evaluates to 2
```

let*: the "sequential" let

In the case of the `let*` form, the scope of a variable also includes the expressions that follow in the list of bindings and the expressions are evaluated from left to right. So, in the example above, `x` is bound before `y`.

letrec: the "recursive" let

In the case of the `letrec` form, the scope of a variable includes all the expressions in the list of bindings, which is particularly useful for recursive procedure definitions.

Quoting

There is a syntax form that is used to **prevent** evaluation: `(quote <expr>)`. `<expr>` is left unevaluated.
Shorthand notation: `'<expr>`.
E.g.,
`(quote (1 2 3))` is a list, without the quote raises an error.
`(quote (+ 2 3))` is another list, without the quote is 5.

Conditional forms

if

```
(if <condition> <then-part> <else-part>)
```

when

```
(when <condition> <then-part>)
```

unless

```
(unless <condition> <else-part>)
```

cond

```
(define n 2)
(cond
    ((= n 0) "zero")
    ((= n 1) "one")
    ((= n 2) "two")
    (else "other"))
    ; => "two"
```

case

```
(define n 2)
(case n
    ((0 1) "small")
    ((2)   "medium")
    ((3 4) "large")
    (else  "other"))
    ; => "medium"
```

Logical forms

and

The `and` form accepts any number of subexpressions, evaluating them from left to right until one returns `#f`. It returns the value of the last expression that was evaluated.

```
> (define n -4)
> (and (>= n -10) (<= n 10) (* n n))
16
> (and (> n 0) (sqrt n)) ;; sqrt not called
#f
```

or

The `or` form accepts any number of subexpressions, evaluating them from left to right until one returns a value other than `#f`. It returns the value of the last expression that was evaluated.

```
> (define n -4)
> (or (odd? n) (positive? n))
#f
> (or (< n 0) (sqrt n)) ;; sqrt not called
#t
```

Sequence of operations: begin

If we are writing a block of procedural code, we can use the `begin` construct.

```
(begin
    (op_1 ...)
    (op_2 ...)
    ...
    (op_n ...))
```

Every `op_i` is evaluated in order, and the value of the `begin` block is the value obtained by the last expression.

Definitions

The `define` form allows defining global variables and procedures.
General syntax:

```
(define <name> <what>)
```

Examples:

```
;; VARIABLES
(define x 12)
(define y #(1 2 3))

;; PROCEDURES
(define cube (lambda (x) (* x x x))) ; (cube 3) => 27

;; COMPACT VARIANT FOR PROCEDURES
(define (cube x) (* x x x))
```

Assignments

`set!` allows changing the value bound to a variable.

```
(begin
    (define x 23)
    (set! x 42)
    x) ; => 42
```

Lists and vectors

There are two types for representing finite sequences of arbitrary values: lists and vectors. Lists allow quickly accessing, adding and removing elements at the front. Vectors are fixed size sequences and offer constant time indexing of any element. Elements of lists and vectors can be of different types.

Lists

Lists are usually constructed with the `list` and `cons` procedures, and elements are accessed with the `car` and `cdr` procedures. Their written representation is `(<element1> <element2> ...)`. A list `(1 2 3)` is stored as `(1 . (2 . (3 . ())))`. `()` is the empty list. List constants in code must be prefixed by the single quote character `'` to avoid treating them as a procedure call.
Some operations on lists:

```
> (define lst (list 1 2))
> lst
(1 2)
> (cons 1 2)           ;; build a pair
(1 . 2)
> (cons 1 '(2))        ;; '(2) = (2 . ())
(1 2)
> (cons 0 lst)         ;; add to front
(0 1 2)
> (append lst '(3 4))  ;; concatenation
(1 2 3 4)
> (car lst)            ;; get first
1
> (cdr lst)            ;; remove first
(2)
> (cdr (cdr lst))      ;; remove both
()
> (length lst)         ;; get length
2
> (member 2 '(1 2 3))  ;; check membership
'(2 3)
```

Vectors

Vectors are usually constructed with the `vector` and `make-vector` procedures, and elements are accessed with the `vector-ref` and `vector-set!` procedures. Their written representation is `#(<element1> <element2> ...)`.
Some operations on vectors:

```
> (define v (make-vector 5 42))
> v
#(42 42 42 42 42)
> (vector-set! v 2 #t)  ;; mutation
> v
#(42 42 #t 42 42)
> (vector-ref v 2)      ;; indexing
#t
> (vector-length v)     ;; get length
5
```

Vectors' mutability

Since constants reside in read-only memory (i.e. regions of the heap explicitly marked to prevent modifications), **literal constants (e.g. the vector #(1 2 3)) are immutable**. If you need a **mutable vector**, use the constructor `(vector 1 2 3)`. Mutation, when possible, is achieved through "bang procedures", e.g. `(vector-set! v 0 "hi")`.

Procedures application

`apply` can be used to apply a procedure to a list of elements.

```
(apply + '(1 2 3 4)) ; => 10
```

Example: minimum of a list

Version 1

```
(define (minimum L)
  (let ((x (car L)) (xs (cdr L)))
    (if (null? xs)  ; is xs = ()?
        x           ; then return x
        (minimum    ; else : recursive call
          (cons
            (if (< x (car xs))
                x
                (car xs))
            (cdr xs))))))
(minimum '(11 -3 2 3 8 -15 0)) ; = > -15
```

Version 2 (variable number of arguments)

```
(define (minimum x . rest)
  (if (null? rest)    ; is rest = ()?
      x               ; then return x
      (apply minimum  ; else : recursive call
        (cons
          (if (< x (car rest))
              x
              (car rest))
          (cdr rest)))))
(minimum 11 -3 2 3 8 -15 0) ; = > -15
```

Loops: the named let

```
(let label ((x 0)) ; variables used in the loop
  (when (< x 10)
    (display x)
    (newline)
    (label (+ x 1)))) ; x++
```

Like with `begin`, the value is the one obtained by the last expression.

Tail recursion

Every Scheme implementation is required to be properly tail recursive. A procedure is called tail recursive if its recursive call is "at the tail", i.e. it is the last operation performed. Tail recursive procedures can be optimized to avoid stack consumption.

Example: factorial

```
;; NOT TAIL RECURSIVE
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))

;; TAIL RECURSIVE
(define (fact x)
  (define (fact-tail x accum) ; local proc
    (if (= x 0)
        accum
        (fact-tail (- x 1) (* x accum))))
  (fact-tail x 1))
```

Equivalence predicates

A predicate is a procedure that returns a Boolean. Its name usually ends with `?` (e.g. `null?`)

- `eq?` tests if two objects are the same (good for symbols)
- `eqv?` like `eq?`, but also checks numbers
- `equal?` predicate is `#t` iff the (possibly infinite) unfoldings of its arguments into regular trees are equal as ordered trees

Loops on lists: for-each

```
(for-each (lambda (x) (displayln x))
  '(this is it))
```

for-each for vectors

We can define a similar procedure working with vectors.

```
(define (vector-for-each body vect)
  (let ((max (- ( vector-length vect) 1)))
    (let loop ((i 0))
      (body (vector-ref vect i))
      (when (< i max)
        (loop (+ i 1))))))
```

Structs

It is possible to define new types, through `struct`.

```
(struct being (
  name               ; name is immutable
  (age #: mutable)   ; flag for mutability
  ))
```

A number of related procedures are automatically created, e.g. the constructor `being` and a predicate to check if an object is of this type: `being?` in this case. Also accessors (and setters for mutable fields) are created.

Structs example

We can define the following procedures:

```
(define (being-show x)
  (display (being-name x))
  (display " (")
  (display (being-age x))
  (display ")"))

(define (say-hello x)
  (if (being? x) ;; check if it is a being
    (begin
      (being-show x)
      (display ": my regards.")
      (newline))
    (error "not a being" x)))
```

Example usage:

```
(define james (being "James" 58))
    (say-hello james)
        ;; => James (58): my regards.
    (set-being-age! james 60) ; a setter
    (say-hello james)
        ;; => James (60): my regards.
```

Clearly it is not possible to change its name.

Structs and inheritance

Structs can inherit:

```
(struct may-being being     ; being is the father
    ((alive? #:mutable)))   ; to be or not to be
```

This being can be killed:

```
(define (kill! x)
    (if (may-being? x)
        (set-may-being-alive?! x #f)
        (error "not a may-being" x)))
```

Dead being are usually untalkative:

```
(define (try-to-say-hello x)
    (if (and
        (may-being? x)
        (not (may-being-alive? x)))
    (begin
        (display "I hear only silence.")
        (newline))
    (say-hello x)))
```

Now we create:

```
(define john (may-being "John" 77 #t))
(say-hello john)
; => John (77): my regards.
```

Note that John is also a being. And destroy:

```
(kill! john)
(try-to-say-hello john)
; => I hear only silence.
```

Closures

A closure is a function together with a referencing environment for the non-local variables of that function. I.e. a function object that "closes" over its visible variables. E.g.,

```
(define (make-adder n)
    (lambda (x)
        (+ x n)))
```

It returns an object that maintains its local value of n.

```
(define add5 (make-adder 5))
(define sub5 (make-adder -5))
(= (add5 5) (sub5 15)) ; => #t
```

Here is a simple application, a closure can be used as an iterator:

```
(define (iter-vector vec)
    (let ((cur 0) (top (vector-length vec)))
        (lambda ()
            (if (= cur top)
                '<<end>>
                (let ((v (vector-ref vec cur )))
                    (set ! cur (+ cur 1))
                    v))))))

(define i (iter-vector #(1 2)))
(i) ; => 1
(i) ; => 2
(i) ; => '<<end>>
```

Higher order functions

Map

$map(f, (e_1, e_2, \dots, e_n)) = (f(e_1), f(e_2), \dots, f(e_n))$

Filter

$filter(p, (e_1, e_2, \dots, e_n)) = (e_i | 1 \leq i \leq n, p(e_i))$

Left and right fold

Let $\bullet$ be a binary operation.

$fold_{left}(\bullet, \iota, (e_1, e_2, \dots, en)) = (e_n \bullet (en_1 \bullet \dots (e_1 \bullet \iota)))$

$fold_{right}(\bullet, \iota, (e_1, e_2, \dots, en)) = (e_1 \bullet (e_2 \bullet \dots (e_n \bullet \iota)))$

Examples

```
(map (lambda (x) (+ 1 x)) '(0 1 2))
; => (1 2 3)
(filter (lambda (x) (> x 0)) '(10 -11 0))
; => (10)
(foldl string-append ""
'("una" " " "bella" " " "giornata"))
; => "giornata bella una"
(foldl cons '() '(1 2 3))
; => (3 2 1)
(foldr cons '() '(1 2 3))
; => (1 2 3)
(foldl * 1 '(1 2 3 4 5 6)) ; i.e. factorial
; = > 720
```

Folds implementation

```
(define (fold-left f i L)
    (if (null? L)
        i
        (fold-left f (f (car L) i) (cdr L))))

(define (fold-right f i L)
    (if (null? L)
        i
        (f (car L) (fold-right f i (cdr L)))))
```

`foldl` is tail recursive, while `foldr` isn't. By saving in a closure the code to be performed after the recursive call, it is possible to make `foldr` tail recursive too. However, although not using the stack, this version builds an even bigger function (the closure one) in the heap. Hence, **the tail recursive version is not more efficient**.

```
(define (fold-right-tail f i L)
    (define (fol d-right-tail-h f i L out)
        (if (null? L)
            (out i)
            (fold-right-tail-h f i
                (cdr L)
                (lambda (x)
                (out (f (car L) x))))))
    (fold-right-tail-h f i L (lambda (x) x)))
```

Macros

General syntax:

```
(define-syntax <name>
    (syntax-rules (<keyword_1> ... <keyword_n>)
        ((<pattern_1>) (<expansion_1>))
        ...
        ((<pattern_n>) (<expansion_n>))))
```

The underscore (_) is used to represent `<name>` in the pattern. The `...` keyword, differently from the above example, where it is freely used with the meaning of "and so forth", is used to match sequences of items.

Hygiene

Scheme macros are *hygienic*. This means that symbols used in their definitions are actually replaced with special symbols not used anywhere else in the program. Therefore it is impossible to have name clashes when the macro is expanded.

While

```
(define-syntax while
    (syntax-rules () ; no other needed keywords
        ((_ condition body ...) ; pattern P
            (let loop () ; expansion of P
                (when condition
                    (begin
                        body ...
                        (loop)))))))
```

While loop example

```
(define (fact-with-while n)
    (let ((x n) (accum 1))
        (while (> x 0)
            (set! accum (* x accum))
            (set! x (- x 1)))
        accum))
```

let* as a recursive macro

```
(define-syntax my-let*
    (syntax-rules ()
        ;; base (= only one variable)
        ((_ ((var val)) istr ...)
        ((lambda (var) istr ...) val))

        ;; more than one
        ((_ ((var val) . rest) istr ...)
        ((lambda (var) (my-let* rest istr ...))
            val))))
```

let as a macro

```
(define-syntax my-let
    (syntax-rules ()
        ((_ ((var expr) ...) body ...)
        ((lambda (var ...) body ...) expr ...))))
```

NB The first `...` in the pattern is used to match a sequence of pairs `(var expr)`, but in the expansion the first `...` gets only the `var` elements, while the last `...` gets only the `expr` elements.

Continuations

A continuation is an abstract representation of the control state of a program.
In practice, it is a data structure used to represent the state of a running program.
The **current continuation** is the continuation that, from the perspective of running code, would be derived from the current point in a program execution.
`call-with-current-continuation` (or `call/cc`) accepts a procedure with one argument, to which it passes the current continuation, implemented as a closure.
The argument of `call/cc` is also called an **escape procedure**. The escape procedure can then be called with an argument that becomes the result of `call/cc`. This means that the escape procedure abandons its own continuation, and reinstates the continuation of `call/cc`.
In practice: we save/restore the call stack.
NB An escape procedure has unlimited extent: if stored, it can be called after the continuation has been invoked, also multiple times.

call/cc examples

Example 1

```
(+ 3
    (call/cc
        (lambda (exit)
            (for-each (lambda (x)
                (when (negative? x)
                    (exit x)))
                '(54 0 37 -3 245 19))
            10))) ; => 0
```

Example 2

```
(define saved-cont #f) ; place to save k

(define (test-cont)
    (let ((x 0))
        (call/cc
            (lambda (k) ; k contains the continuation
                (set! saved-cont k))) ; here is saved

        ;; this *is* the continuation
        (set! x (+ x 1))
        (display x)
        (newline)))
```

At the REPL:

```
(test-cont)     ;; => 1
(saved-cont)    ;; => 2
(define other-cont saved-cont)
(test-cont)     ;; => 1 (here we reset saved-cont)
(other-cont)    ;; => 3 (other is still going...)
(saved-cont)    ;; => 2
```

Example 3: a for with break

```
;; DEFINITION
(define-syntax For
    (syntax-rules (from to break: do)
        ((_ var from min to max break:
            br-sym do body ...)
        (let* ((min1 min) (max1 max)
            (inc (if (< min1 max1) + -)))
            (call/cc (lambda (br-sym)
                (let loop ((var min1))
                    body ...
                    (unless (= var max1)
                        (loop (inc var 1))))))))))

;; USAGE
(For i from 1 to 10 break: get-out
    do (displayln i)
        (when (= i 5)
            (get-out)))
```

Example 4: throw/catch exception

```
;; HANDLERS
; stack for installed handlers
(define *handlers* (list))

(define (push-handler proc)
    (set ! *handlers* (cons proc *handlers*)))

(define (pop-handler)
    (let ((h (car *handlers*)))
        (set! *handlers* (cdr *handlers*))
        h))

;; THROW
; throw: if there is a handler, we pop and call it
; otherwise we raise an error
(define (throw x)
    (if (pair? *handlers*)
        ((pop-handler) x)
        (apply error x)))

;; TRY-CATCH
(define-syntax try
    (syntax-rules (catch)
        ((_ exp1 ...
            (catch what hand ...))
        (call/cc (lambda (exit)
            ; install the handler
            (push-handler (lambda (x)
                (if (equal? x what)
                    (exit
                        (begin
                            hand ...))
                    (throw x))))
            (let ((res ;; evaluate the body
                (begin exp1 ...)))
                ; ok: discard the handler
                (pop-handler)
                res))))))

;; EXAMPLE WITH THROW-CATCH
(define (foo x)
    (display x) (newline)
    (throw "hello"))

(try
    (display "Before foo")
    (newline)
    (foo "hi!")
    (display "After foo") ; unreached code
    (catch "hello"
        ; this is the handler block
        (display "I caught a throw.") (newline)
        #f))

;; CATCH, MACRO-EXPANDED
(call/cc
    (lambda (exit)
        (push-handler
            (lambda (x)
                (if (equal? x "hello")
                    (exit (begin
                        (display "I caught a throw.")
                        (newline)
                        #f))
                    (throw x))))
        (let ((res (begin
                        (display "Before foo")
                        (newline)
                        (foo "hi!")
                        (display "After foo"))))
            (pop-handler)
            res)))
```

Closures vs OO

It is possible to use closures to do some basic OO programming. **The main idea is to define a procedure which assumes the role of a *class*. This procedure, when called, returns a *closure* that works like an *object*.** It works by implementing information hiding through the "enclosed" values of the closure. **Access to the state is through *messages* to a function that works like a *dispatcher*.**

Closures as objects

```
(define (make-simple-object)
    (let ((my-var 0)) ; attribute

    ;; methods:
    (define (my-add x)
        (set! my-var (+ my-var x))
        my-var)
    (define (get-my-var)
        my-var)
    (define (my-display)
        (newline)
        (display "my Var is: ")
        (display my-var)
        (newline))

    ;; DISPATCHER
    (lambda (message . args)
        (apply (case message
                    ((my-add) my-add)
                    ((my-display) my-display)
                    ((get-my-var) get-my-var)
                    (else (error "Unknown Method!")))
            args))))

;; EXAMPLE USAGE
; NB make-simple-object returns a closure
; which contains the dispatcher
(define a (make-simple-object))
(define b (make-simple-object))
(a 'my-add 3)   ; => 3
(a 'my-add 4)   ; => 7
(a 'get-my-var) ; => 7
(b 'get-my-var) ; => 0
(a 'my-display) ; => My Var is: 7
```

Inheritance by delegation

```
(define (make-son)
    (let ((parent (make-simple-object)) ; inheritance
        (name "an object"))

        (define (hello)
            "hi!")
        (define (my-display)
            (display "My name is ")
            (display name)
            (display " and")
            (parent 'my-display ))

        (lambda (message . args)
            (case message
                ((hello) (apply hello args))
                ;; overriding
                ((my-display) (apply my-display args))
                ;; parent needed
                (else (apply parent
                        (cons message args)))))))

;; EXAMPLE USAGE
(define c (make-son))
(c 'my-add 2)
(c 'my-display) ; => My name is an object and
                ; my Var is: 2
(display (c 'hello)) ; => hi!
```

Prototype-based Object Orientation

There are no classes: new objects are obtained by **cloning** and modifying existing object.
We will see here how to implement it on top of Scheme, using **hash tables** as the main data structure.

```
;; An object is implemented with a hash table
(define new-object make-hash)
(define clone hash-copy)
;; keys are attribute/method names

;; Utilities (syntactic sugar)
(define-syntax !!   ;; setter
    (syntax-rules ()
        ((_ object msg new-val)
        (hash-set! object 'msg new-val))))

(define-syntax ??   ;; reader
    (syntax-rules ()
        ((_ object msg)
        (hash-ref object 'msg))))

(define-syntax ->   ;; send message
    (syntax-rules ()
        ((_ object msg arg ...)
        ((hash-ref object 'msg) object arg ...))))

;; EXAMPLE
;; Object and methods definition
(define Pino (new-object))
(!! Pino name "Pino") ;; slot added
(!! Pino hello
    (lambda (self x) ;; method added
        (display (?? self name))
        (display ": hi , ")
        (display (?? x name ))
        (display "!")
        (newline)))
;; Additional methods
(!! Pino set-name
    (lambda (self x)
        (!! self name x)))

(!! Pino set-name-&-age
    (lambda (self n a)
        (!! self name n)
        (!! self age a)))

;; Clone
(define Pina (clone Pino))
(!! Pina name "Pina")

;; USAGE
(-> Pino hello Pina)    ; Pino: hi, Pina!
(-> Pino set-name "Ugo")
(-> Pina set-name-&-age "Lucia" 25)
(-> Pino hello Pina)    ; Ugo: hi, Lucia!

;; INHERITANCE (by delegation)
(define (deep-clone obj)
    (if (not (hash-ref obj '<<parent>> #f))
        (clone obj)
        (let* ((cl (clone obj))
            (par (?? cl <<parent>>)))
        (!! cl <<parent>> (deep-clone par)))))

(define (son-of parent)
    (let ((o (new-object)))
        (!! o <<parent>> (deep-clone parent))
        o))

;; DISPATCHING
(define (dispatch object msg)
    (if (eq? object 'unknown)
        (error "Unknown message" msg)
        (let ((slot (hash-ref
                    object msg 'unknown)))
            (if (eq? slot 'unknown)
                (dispatch (hash-ref object
                        '<<parent>> 'unknown) msg)
                slot))))

;; We now have to modify ?? and -> for dispatching
(define-syntax ?? ;; reader
    (syntax-rules ()
        ((_ object msg)
        (dispatch object 'msg))))

(define-syntax -> ;; send message
    (syntax-rules ()
        ((_ object msg arg ...)
        ((dispatch object 'msg) object arg ...))))

;; EXAMPLE
(define Glenn (son-of Pino))
(!! Glenn name "Glenn")
(!! Glenn age 50)
(!! Glenn get-older
    (lambda (self)
        (!! self age (+ 1 (?? self age)))))

(-> Glenn hello Pina) ; Glenn: hi, Lucia!
(-> Glenn ciao) ; error: Unknown message
(-> Glenn get-older) ; Glenn is now 51
```

Past exams

2023/09/12

#MACROS

Text

1. Design a construct to define multiple functions with the same number of arguments at the same time. The proposed syntax is the following:
   (multifun <list of function names> <list of parameters> <list of bodies>).
   E.g.
   (multifun (f g) (x) ((+ x x x) (x x)))*
   defines the two functions *f* with body *(+ x x x)* and *g* with body *(* x x)*, respectively.
2. Would be possible to define something similar, but using a procedure and lambda functions instead of a macro? If yes, do it; if no, explain why.

Solution

1.
```scheme
(define-syntax multifun
    (syntax-rules ()
        ((_ (f) (x ...) (b))
            (define (f x ...) b))
        ((_ (f . fs) (x ...) (b . bs))
            (begin
                (define (f x ...) b)
                (multifun fs (x ...) bs)))))
```

2. Of course we can use a list of lambda functions instead of the list of bodies (alas, we need to replicate the list of parameters on each body). The main problem is that we cannot bind the top-level function names from inside a procedure, so the answer is no.

2023/07/03

#MACROS #LISTS

Text

Define a `let**` construct that behaves like the standard `let*`, but gives to variables provided without a binding the value of the last defined variable. It also contains a default value, stated by a special keyword `def:`, to be used if the first variable is given without binding. For example:

```scheme
(let** def: #f
    (a (b 1) (c (+ b 1)) d (e (+ d 1)) f)
    (list a b c d e f))
```

should return `'(#f 1 2 2 3 3)`, because `a` assumes the default value `#f`, while `d` = `c` and `f` = `e`.

Solution

```scheme
(define-syntax let**
(syntax-rules (def:)
    ; base case to default
    ((_ def: d (var) body ...)
        ((lambda (var) body ...) d))

    ; base case with binding
    ((_ def: d ((var val)) body ...)
        ((lambda (var) body ...) val))

    ; recursive case with binding
    ((_ def: d ((var val) . rest) body ...)
        ((lambda (var)
            (let** def: val rest body ...)) val))

    ; recursive case to default
    ((_ def: d (var . rest) body ...)
        ((lambda (var)
            (let** def: d rest body ...)) d))))

;; NB The order of the possible matching matters
```

2023/06/13

#FOLD #LISTS

Text

Write a function, called *fold-left-right*, that computes both *fold-left* and *fold-right*, returning them in a pair. Very important: the implementation must be *one-pass*, for efficiency reasons, i.e. it must consider each element of the input list only once; hence it is not correct to just call Scheme's fold-left and -right.
Example: `(fold-left-right string-append "" '("a" "b" "c"))` is the pair `("cba" . "abc")`.

Solution

```scheme
(define (fold-left-right f i l)
  (let loop ((left i)
             (right (lambda (x) x))
             (xs l))
    (if (null? xs)
        (cons left (right i))
        (loop (f (car xs) left)
              (lambda (x)
                (right (f (car xs) x)))
              (cdr xs)))))
```

2023/02/15

#MACROS #CALLCC

Text

Consider the following *For* construct, as defined in class:

```scheme
(define-syntax For
    (syntax-rules (from to break: do)
        ((_ var from min to max break:
            break-sym do body ...)
        (let* ((min1 min) (max1 max)
            (inc (if (< min1 max1) + -)))
            (call/cc (lambda (break-sym)
                (let loop ((var min1))
                    body ...
                    (unless (= var max1)
                        (loop (inc var 1))))))))))
```

Define a fix to the above definition, to avoid to introduce in the macro definition the special break symbol *break-sym*, by providing a construct called *break*. E.g.

```scheme
(For i from 1 to 10
    do
    (displayln i)
    (when (= i 5)
        (break #t)))
```

will return *#t* after displaying the numbers from 1 to 5.

Solution

```scheme
(define exit-store '())
(define (break v)
    ((car exit-store) v))

(define-syntax For+
 (syntax-rules (from to do)
  ((_ var from min to max
   do body ...)
    (let* ((min1 min)
           (max1 max)
           (inc (if (< min1 max1) + -)))
     (let ((v (call/cc (lambda (break)
                        (set! exit-store
                         (cons break exit-store))
                        (let loop ((var min1))
                         body ...
                         (unless (= var max1)
                          (loop (inc var 1))))))))
          (set! exit-store (cdr exit-store))
          v)))))
```

2023/01/25

#MACROS #CALLCC

Text

We want to implement a *for-each/cc* procedure which takes a condition, a list and a body and performs a for-each. The main difference is that, when the condition holds for the current value, the continuation of the body is stored in a global queue of continuations. We also need an auxiliary procedure, called *use-cc*, which extracts and call the oldest stored continuation in the global queue, discarding it.
E.g. if we run:

```scheme
(for-each/cc odd?
    '(1 2 3 4)
    (lambda (x) (displayln x)))
```

two continuations corresponding to the values 1 and 3 will be stored in the global queue.
Then, if we run: `(use-scc)`, we will get on screen:
2
3
4

Solution

```scheme
(define queue '()) ;; global queue

;; appends x to the end of the queue
(define (enqueue x)
  (set! queue (append queue (list x))))
```

```
(define (dequeue)
  (unless (null? queue)
    (let ((x (car queue)))
      (set! queue (cdr queue))
      (x)))) ;; return the dequeued value

(define-syntax for-each/cc
  (syntax-rules ()
    ((_ condition list body)
     (for-each (lambda (x)
                 (call/cc (lambda (c)
                            (when (condition x)
                              (enqueue c))
                            (body x))))
               list))))

(define (use-cc) (dequeue))
```

2022/09/01

#MACROS #CALLCC

Text

We want to implement a version of call/cc, called *store-cc*, where the continuation is called only once and it is implicit, i.e. we do not need to pass a variable to the construct to store it. Instead, to run the continuation, we can use the associated construct *run-cc* (which may take parameters). The composition of *store-cc* must be managed using in the standard last-in-first-out approach.
E.g. if we run:

```
(define (test)
    (define x 0)
    (store-cc
        (displayln "here")
        (set! x (+ 1 x)))
    (displayln x)
    (set! x (+ 1 x))
    x)
(test)
```

we will get: `here`
`1`
`2`
and if we call `(run-cc)`
we get:
`2`
`3`
and the continuation is discarded.

Solution

```
(define *stored-cc* '())

(define-syntax store-cc
    (syntax-rules ()
        ((_ body ...)
         (call/cc (lambda (k)
                    (set! *stored-cc*
                        (cons k *stored-cc*))
         body ...)))))

(define (run-cc . v)
    (let ((k (car *stored-cc*)))
    (set! *stored-cc* (cdr *stored-cc*))
    (apply k v)))

(define (test)
    (define x 0)
    (store-cc
        (displayln "here")
        (set! x (+ 1 x)))
    (displayln x)
    (set! x (+ 1 x))
    x)
```

2022/07/06

#CLOSURE #OOP

Text

Consider the technique “closures as objects” as seen in class, where a closure assumes the role of a class. In this technique, the called procedure (which works like a class in OOP) returns a closure which is essentially the dispatcher of the object.
Define the *define-dispatcher* macro for generating the dispatcher in an automatic way, as illustrated by the following example:

```
(define (make-man)
    (let ((p (make-entity)) (name "man"))
        (define prefix+name
            (lambda (prefix)
                (string-append prefix name)))
    (define change-name
        (lambda (new-name)
            (set! name new-name)))
    (define-dispatcher methods:
        (prefix+name change-name) parent: p)))
```

where `p` is the parent of the current instance of class man, and `make-entity` is its constructor.
If there is no inheritance (or it is a base class), *define-dispatcher* can be used without the `parent: p` part.
Then, an instance of class `man` can be created and its methods can be called as follows:

```
> (define carlo (make-man))
> (carlo 'change-name "Carlo")
> (carlo 'prefix+name "Mr. ")
"Mr. Carlo"
```

Solution

```
(define (unknown-method ls)
  (error "Unkwown method" (car ls)))

(define-syntax define-dispatcher
  (syntax-rules (methods: parent:)
    ((_ methods: (mt ...) parent: p)
     (lambda (message . args)
       (case message
         ((mt) (apply mt args))
         ...
         (else (apply p (cons message args))))))
    ((_ methods: mts)
     (define-dispatcher methods:
       mts parent: unknown-method))))
```

2022/06/16

#FOLD #LAMBDA

Text

Define a *list-to-compose* pure function, which takes a list containing functions of one argument and returns their composition.
E.g. *(list-to-compose (list f g h))* is the function *f(g(h(x)))*.

Solution

```
(define (list-to-compose lst)
(lambda (x)
    (foldr (lambda (y acc)
        (y acc)) x lst)))
```

2022/02/10

#CLOSURE

Text

Consider the following code:

```
(define (r x y . s)
    (set! s (if s (car s) 1))
    (lambda ()
        (if (< x y)
            (let ((z x))
                (set! x (+ s x))
                z)
            y)))
```

1. What can we use *r* for? Describe how it works and give some useful examples of its usage.
2. Does it make sense to create a version of *r* without the *y* parameter? If the answer is yes, implement such version; if no, explain why.

Solution

It is a generator implemented as a closure, which returns the numbers from x to y with step s (+1 if s is not defined). When y is reached, it returns it indefinitely.
Yes, y is the upper limit and we could drop it:

```
(define (r1 x . s)
    (set! s (if s (car s) 1))
    (lambda ()
        (let ((z x))
            (set! x (+ s x))
            z)))
```

2022/01/21

#MACROS

Text

Define a new construct called *block-then* which creates two scopes for variables, declared after the scopes, with two different binding. E.g. the evaluation of the following code:

```
    (block
        ((displayln (+ x y))
        (displayln (* x y))
        (displayln (* z z)))
    then
        ((displayln (+ x y))
        (displayln (* z x)))
    where (x <- 12 3)(y <- 8 7)(z <- 3 2))
```

should show on the screen:

```
20
96
9
10
6
```

Solution

It is a generator implemented as a closure, which returns the numbers from x to y with step s (+1 if s is not defined). When y is reached, it returns it indefinitely.

Yes, y is the upper limit and we could drop it:

```
    (define-syntax block
        (syntax-rules (then where <-)
            (
                (_ (e1 ...) then (e2 ...) where
                    (var <- val1 val2) ...)
                (begin
                    (let ((var val1) ...) e1 ...)
                    (let ((var val2) ...) e2 ...))
            )
        )
    )
```

Made by Antonio Sgarlata for the *Principles of Programming Languages* course held at Politecnico di Milano.
Template by Dave Richeson, Dickinson College,
http://divisbyzero.com/