Principle of Programming Languages

Scheme

Lambda

Lambdas are unnamed procedures

```
(lambda (x y) ; this is a comment
    (+ (* x x) (* y y)))
```

Example usage:

```
((lambda (x y) (+ (* x x) (* y y))) 2 3) ;
    => 23
```

Let

Scope is **static** let example:

```
(let ((x 2)
      (y 3))
    ;here go the code using x,y
)
```

let binds variables "in parallel":

```
(let ((x 1)
      (y 2))
    (let ((x y)
          (y x))
        ;y = 1, x = 2
    )
)
```

Quoting

Syntax form to prevent evaluation

```
(quote <expr>)
;equivalent to
'<expr>
```

<expr> is left unevaluated

```
(quote (1 2 3)) ;is a list
(quote (+ 1 2)) ;is another list
```

quasiquote (‘) and *unquote* (,) are used for partial evaluation

```
'(1 2 3) ;is a list (1 2 3)
‘(1 ,(+ 1 1) 2) ;is another list (1 2 3)
```

Eval

```
eval '(+ 1 2 3) ;is 6
```

Begin

Use **begin** to write block of procedural code:

```
(begin
    (op-1 ...)
    (op-2 ...)
    ...
    (op-n ...)
)
```

Every op-i is evaluated in order and the value of the block is the value obtained by the last expression

Definitions

To create top-level bindings there is **define**:

```
(define <name> <what>)
;example
(define x 12)
(define y #(1 2 3))
```

Defining procedures:

```
(define cube (lambda (x) (* x x x)))
(cube 3) ; = 27
```

Short notation to define procedures:

```
(define (cube x) (* x x x))
(cube 3) ; = 27
```

Assignment

Use **set!** for assignement:

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

List

Lists are memorized as concatenated pairs, a pair (written (x . y), also called a cons node) consists of two parts:

- **car** (i.e. x) aka Content of the Address Register (take the first element of a list)
- **cdr** (i.e. y) aka Content of the Data Register (take the list without the first element)

A list (1 2 3) is stored like this (1 . (2 . (3 . ())))
() is the empty list
The two procedures **car** and **cdr** are used as accessors
member is used to check if a list contains a value:

```
(member 2 '(1 2 3)) ; => '(2 3)
```

Define a procedure with variable number if argument

```
(define (x . y) y)
(x 1 2 3) ; = > '(1 2 3)
```

apply can be used to apply procedure to list of elements:

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

To build pair we can use **cons**

```
(cons 1 2) ;   = (1 . 2)
(cons 1 '(2)) ; = (1 2)
```

Example, find minimum of a list:

```
(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
```

Variant with variable number of argument:

```
(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 xs)
                )
                (cdr xs)
            )
        )
```

```
    )
)
(minimum 11 -3 2 3 8 -15 0) ; = > -15
```

Loop

Example:

```
(let label ((x 0))
    (when (< x 10)
        (display x)
        (newline)
        (label (+ x 1))
    )
)
```

We can use as many variables as we like, value obtained by the last expression.

Tail recursion

Example **not tail recursive**:

```
(define (factorial n)
    (if (= n 0)
        1
        (* n factorial (- n 1))
    )
)
```

Example **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)
)
```

To avoid stack consumption it can be optimize (indeed, the previous tail call is translated in the following low-level code)

```
(define (fact-low-level n)
    (define x n)
    (define accum 1)
    (let loop () ; see this as the "loop"
        label
        (if (= x 0)
            accum
            (begin
                (set! accum (* x accum))
                (set! x (- x 1))
                (loop) ; jump to "loop"
            )
        )
    )
)
```

A more idiomatic way of writing it is the following:

```
(define (fact-low-level n)
    (let loop ((x n)
               (accum 1))
        (if (= x 0)
            accum
            (loop (- x 1)(* accum x))
        )
    )
)
```

But note that this looks like a tail call. . . (In reality, the named let is translated into a local recursive function. If tail recursive, when compiled it becomes a simple jump.)

for-each

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

For vector:

```
(vector-for-each (lambda (x)
                   (display x)(newline))
                 #(this is it)
)
```

Here is the definition:

```
(define (vector-for-each body vect)
    (let ((max (- (vector-leght vect) 1)))
        (let loop ((i 0))
            (body (vector-ref vect i)) ;
                vect[i] in C
            (when (< i max)
                (loop (+ x 1))
            )
        )
    )
)
```

Equivalence

- A predicate is a procedure that returns a Boolean. Its name usually ends with ? (e.g. null?)
- = is used only for numbers
- there are:
  - **eq?** tests if two objects are the same (good for symbols) *(eq? 'casa 'casa), but not (eq? "casa" (string-append "ca" "sa")), (eq? 2 2) is unspecified*
  - **eqv?** like eq?, but checks also numbers
  - **equal?** predicate is true iff the (possibly infinite) unfoldings of its arguments into regular trees are equal as ordered trees. *(equal? (make-vector 5 'a)(make-vector 5 'a)) is true*

Case and Cond

- case:

```
(case (car '(c d))
    ((a e i o u)   'vowel)
    ((w y)         'semivowel)
    (else          'constant)
)   ; => constant
```

- cond:

```
(cond   ((> 3 3) 'greater)
        ((< 3 3) 'less)
        (else    'equal)
)   ; => equal
```

Storage model and mutability

- Variables and object implicitly refer to locations or sequence of locations in memory (usually the heap)
- Scheme is garbage collected: every object has unlimited extent – memory used by objects that are no longer reachable is reclaimed by the GC
- Constants reside in read-only memory (i.e. regions of the heap explicitly marked to prevent modifications), therefore literal constants (e.g. the vector #(1 2 3)) are immutable
- If you need e.g. a mutable vector, use the constructor (vector 1 2 3)
- Mutation, when possible, is achieved through "bang procedures", e.g. (vector-set! v 0 "moose")

Vector are mutable only if created with standard constructor

```
(define (f)
    (let ((x (vector 1 2 3)))
        x
)   )

(define v (f))
(vector-set! v 0 10)
```

```
(display v)(newline) ; => #(10 2 3)
```

Vector are immutable if created with #(x y z)

```
(define (g)
    (let ((x #(1 2 3)))
        x
)   )

(display (g))(newline) ; = > #(1 2 3)
(vector-set! (g) 0 10) ; = > error !
```

List:

- In Racket, lists are immutable (so no set-car!, set-cdr!) - this is different from most Scheme implementations (but it is getting more common)
- There is also a mutable pair datatype, with mcons, set-mcar!, set-mcdr!

Evaluation Strategy

Evaluation strategy: call by object sharing (like in Java): objects are allocated on the heap and references to them are passed by value.

```
(define (test-setting-local d)
    (set! d "Local") ; setting the local d
    (display d)(newline))

(define ob "Global")
(test-setting-local ob) ; = > Local
(display ob)          ; = > Global
```

- It is also often called call by value, because objects are evaluated before the call, and such values are copied into the activation record
- The copied value is not the object itself, which remains in the heap, but a reference to the object
- This means that, if the object is mutable, the procedure may exhibit side effects on it

```
(define (set-my-mutable d)
    (vector-set! d 1 "done")
    (display d))

(define ob1 (vector 1 2 3)) ; i.e. #(1 2 3)
(set-my-mutable ob1)  ; = > #(1 done 3)
(display ob1)         ; = > #(1 done 3)
```

Struct

It is possible to define new types, through **struct**. The main idea is like struct in C, with some differences.

```
(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 e.g., we can define the following procedure:

```
(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 (58): my
    regards.
; it is not possible to change its name
```

Structs can inherit

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

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

(define (try-to-say-hello x)
    (if (and (my-being? x) (not
        (may-being-alive? x)))
        (begin
            (display "I hear only silence.")
            (newline)
        )
        (say-hello x))
)
; Example usage
(define john (may-being "Jhon" 77 #t))
(say-hello jhon)  ; => John (77): my regards.
(kill! jhon)
(try-to-say-hello john) ; => I hear only
    silence.
```

Structs vs Object-Oriented programming

The main difference is in methods vs procedures:

- procedures are external, so with inheritance we cannot redefine/override them
- still, a may-being behaves like a being
- but we had to define a new procedure (i.e. try-to-say-hello), to cover the role of say-hello for a may-being
- structs are called records in the standard.

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 example:

```
(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 > >
```

Some classical higher order functions

- **map**:

$$map(f, (e_1, e_2, \ldots, e_n)) = (f(e_1), f(e_2), \ldots, f(e_n)) \quad (1)$$

- **filter**:

$$filter(p(e_1, e_2, \ldots, e_n)) = (e_i | 1 \leq i \leq n, p(e_i)) \quad (2)$$

- **foldr**:

$$fold_{right}(\circ, \iota, (e_1, e_2, \ldots, e_n)) = (e_1 \circ (e_2 \circ \ldots (e_n \circ \iota))) \quad (3)$$

- **foldl**:

$$fold_{left}(\circ, \iota, (e_1, e_2, \ldots, e_n)) = (e_n \circ (e_{n-1} \circ \ldots (e_1 \circ \iota))) \quad (4)$$

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
```

Example implementation of folds (foldl is tail recursive, while foldr isn't)

```
(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)))))
```

An implementation of tail-recursive foldr (not gain much)

```
(define (fold-right-tail f i L)
    (define (fold-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)))
```

Meta-programming through Macros

While loops, we cannot define it as a procedure But we can usa a macro:

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

`let*` as a 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))))
```

And `let` as a macro:

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

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
- on the other hand, sometime we want name clashes, so these particular cases can be tricky (we will see an example later)

Continuation

- 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
- if the language supports first class functions, it is always possible to refactor code in continuation passing style, where control is passed explicitly in the form of a continuation
- (hint: we saw an example with the tail-recursive fold-right)
- Scheme, unlike many mainstream languages, natively supports continuations:
- 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 (see next example)
- In practice: we save/restore the call stack

Example:

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

An escape procedure has unlimited extent: if stored, it can be called after the continuation has been invoked, also multiple times

```
(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)))

;; USAGE
(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
```

Another example: a for with break

```
(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)))
```

Exceptions

First we need a 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: 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)))
```

Exceptions: 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))))))
```

An 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))
```

Object Oriented programming

It is possible to use closures to do some basic OO programming

```
(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))
        (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:

```
(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      "anobject"))
        ; methods :
        (define (hello)
            "hi!")
        (define (my-display)
            (newline)
            (display "My name is ")
            (display name)
            (display " and")
            (parent 'my-display))
        (lambda (message . args)
            (case message
                ((hello)       (apply hello
                    args))
                ; override
                ((my-display)  (apply
                    my-display args))
                ; parent needed
                (else (apply parent (cons
                    message args)))))))
```

Example usage:

```
(define c (make-son))
(c 'my-add 2)      ; => 2
```

```
(c 'my-display)    ; => My name is an object
    and
                   ;    My Var is : 2
(display (c 'hello)) ; => hi!
```

Prototype-Based Object system

Some hash table function:

```
(hash-ref hash key [failure-result]) -> any
    ;Returns the value for key in hash. If no
    value is found for key, then failure-result
    determines the result
(hash-set! hash key v) -> void? ; Maps key to v
    in hash, overwriting any existing mapping
    for key.
```

An object is implemented with a hash table

```
(define new-object make-hash)
(define clone hash-copy)
```

keys are attribute/method names.
Just for convenience:

```
(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
             ...))))
```

First, we define an object and its methods

```
(define Pino (new-object))
(!! Pino name "Pino") ; slot added
(!! Pino hello
    (lambda (self x)
        (display (?? self name))
        (display ": hi, ")
        (display (?? x name"))
        (display "!")
        (newline)))
(!! Pino set-name
    (lambda (self x)
        (!! self name x)))
(!! Pino set-name-&-age
    (lambda (self n a)
        (!! self name n)
        (!! self age a)))
```

Now create object Pina as clone of Pino

```
(define Pina (clone Pino))
(!! Pina name "Pina")
; example usage
(-> Pino hello Pina)  ; Pino: hi, Pina!
(-> Pino set-name "Ugo")
(-> Pina set-name-&-age "Lucia" 25)
(-> Pino hello Pina)  ; Ugo: hi, Lucia!
```

Proto-OO: Inheriance

```
(define (son-of parent)
    (let ((o (new-object)))
        (!! o <<parent>> parent)
        o))
; Basic dispatching
(define (dispatch object msg)
    (if (eq? object 'unknown)
        (error "Unknown message" message)
        (let ((slot (hash-ref object msg
            'unknown)))
            (if (eq? slot 'unknown)
                (dispatch (hash-ref object
                    '<<parent>> 'unknown) msg)
                slot))))
; 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 usage:

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

Exams

2020-07-17

Define the verbose construct for folding illustrated by the following example. This is a fold-right (− >) with initial value 1 on the list (1 2 3 4 5 6), and the fold function is given in the "exec" part. Of course, < − is used to select fold-left instead of right.
Solution:

```
(define-syntax cobol-fold
   (syntax-rules (direction -> <- data using
       from exec)
      ((_ direction -> from i data d ... (exec
          e ... ) using x y)
        (foldr (lambda (x y) e ...) i '(d
            ...)))
      ((_ direction <- from i data d ... (exec
          e ... ) using x y)
        (foldl (lambda (x y) e ...) i '(d
            ...)))))

(cobol-fold direction -> from 1 data 1 2 3 4
    5 6
     (exec
       (displayln y)
       (+ x y))
       using x y)
```

2020-06-29

Define the construct define-with-types, that is used to define a procedure with type constraints, both for the parameters and for the return value. The type constraints are the corresponding type predicates, e.g. number? to check if a value is a number. If the type constraints are violated, an error should be issued.
Solution:

```
(define-syntax define-with-types
   (syntax-rules (:)
      ((_ (f : tf (x1 : t1) ...) e1 ...)
        (define (f x1 ...)
          (if (and (t1 x1) ...)
            (let ((res (begin
```

```
                e1 ...)))
              (if (tf res)
                res
                (error "bad return type")))
            (error "bad input types"))))))

(define-with-types (add-to-char : integer? (x
    : integer?) (y : char?))
  (+ x (char->integer y)))

(add-to-char 3 #\A)
```

2020-02-07

Implement this new construct: (each-until var in list until pred : body), where keywords are written in boldface. It works like a for-each with variable var, but it can end before finishing all the elements of list when the predicate pred on var becomes true.
Solution:

```
(define-syntax each-until
  (syntax-rules (in until :)
    ((_ x in L until pred : body ...)
      (call/cc
        (lambda (exit)
          (let ((pred-fun (lambda (x)
              pred)))
            (let loop ((xs L))
              (if (null? xs)
                (exit)
                (let ((x (car xs)))
                  (if (pred-fun x)
                    (exit)
                    (begin
                      body ...
                      (loop (cdr
                          xs))))))))))))))
```

2020-01-15

Consider the Foldable and Applicative type classes in Haskell. We want to implement something analogous in Scheme for vectors. Note: you can use the following library functions in your code: vectormap, vector-append.

- Define vector-foldl and vector-foldr.
- Define vector-pure and vector-$<*>$.

Solution:

```
(define (vector-foldr f i v)
  (let loop ((cur (- (vector-length v) 1))
             (out i))
    (if (< cur 0)
        out
        (loop (- cur 1) (f (vector-ref v
            cur) out)))))
(define (vector-foldl f i v)
  (let loop ((cur 0)
             (out i))
    (if (>= cur (vector-length v))
        out
        (loop (+ cur 1) (f (vector-ref v
            cur) out)))))
(define (vector-concat-map f v)
  (vector-foldr vector-append #()
      (vector-map f v)))
(define vector-pure vector)
(define (vector-<*> fs xs)
  (vector-concat-map (lambda (f)
      (vector-map f xs)) fs))
```

2019-09-03

Consider the following code:

```
(define (a-function lst sep)
  (foldl (lambda (el next)
           (if (eq? el sep)
             (cons '() next)
             (cons (cons el (car next))
               (cdr next))))
    (list '()) lst))
```

- 1) Describe what this function does; what is the result of the following call?
  `(a-function '(1 2 nop 3 4 nop 5 6 7 nop nop 9 9 9) 'nop)`
- 2) Modify a-function so that in the example call the symbols nop are not discarded from the resulting list, which must also be reversed (of course, without using reverse).

Solution:

```
a-function returns a list of lists, where
    each list is taken backwards, and sep is
    used for a separator. The resulting list
    is: ((9 9 9) () (7 6 5) (4 3) (2 1))
The modified function is:
(define (another-function lst sep)
  (foldr (lambda (el next)
           (if (eq? el sep)
             (cons (list el) next)
             (cons (cons el (car next))
               (cdr next))))
    (list '()) lst))
```

2019-07-24

Write a functional, tail recursive implementation of a procedure that takes a list of numbers L and two values x and y, and returns three lists: one containing all the elements that are less than both x and y, the second one containing all the elements in the range [x,y], the third one with all the elements bigger than both x and y. It is not possible to use the named let construct in the implementation.
Solution:

```
(define (3-part L v1 v2)
  (define (3-p L v1 v2 r1 r2 r3)
    (if (null? L)
      (list r1 r2 r3)
      (let   ((x (car L))
              (xs (cdr L)))
        (cond
          ((and (< x v1)(< x v2))
            (3-p xs v1 v2 (cons x
                r1) r2 r3))
          ((and (>= x v1)(<= x v2))
            (3-p xs v1 v2 r1 (cons
                x r2) r3))
          ((and (> x v1)(> x v2))
            (3-p xs v1 v2 r1 r2
                (cons x r3)))))))
  (3-p L v1 v2 '() '() '()))
```

2019-06-28

Consider this data definition in Haskell: data Tree a = Leaf a — Branch (Tree a) a (Tree a)
Define an OO analogous of this data structure in Scheme using the technique of "closure as classes" as seen in class, defining the map and print methods, so that:
`(define t1 (Branch (Branch (Leaf 1) -1 (Leaf 2)) -2 (Leaf 3)))`
`((t1 'map (lambda (x) (+ x 1))) 'print)`
should display: `(Branch (Branch (Leaf 2) 0 (Leaf 3)) -1 (Leaf 4))`
Solution:

```
(define (Branch t1 x t2)
  (define (print)
    (display "(Branch ")
    (t1 'print)
    (display " ")
    (display x)
    (display " ")
    (t2 'print)
    (display ")"))
```

```
    (define (map f)
        (Branch (t1 'map f) (f x) (t2 'map
            f)))
    (lambda (message . args)
        (apply
            (case message
                ((print) print)
                ((map) map)
                (else (error "Unknown")))
            args)))

(define (Leaf x)
    (define (print)
        (display "(Leaf ")
        (display x)
        (display ")"))
    (define (map f)
        (Leaf (f x)))
    (lambda (message . args)
        (apply
            (case message
                ((print) print)
                ((map) map)
                (else (error "Unknown")))
            args)))
```

2019-02-08

Define a pure function `multi-merge` with a variable number of arguments (all of them must be ordered lists of numbers), that returns an ordered list of all the elements passed.
It is forbidden to use external sort functions.
E.g. when called like:
`(multi-merge '(1 2 3 4 8) '(-1 5 6 7) '(0 3 8) '(9 10 12))`
it returns: `'(-1 0 1 2 3 3 4 5 6 7 8 8 9 10 12)`
Solution:

```
(define (merge a1 a2)
    (cond ((null? a1) a2)
        ((null? a2) a1)
        (else  (let ((x (car a1))
                     (y (car a2)))
                 (if (< x y)
                     (cons x (merge (cdr a1)
                         a2))
                     (cons y (merge a1 (cdr
                         a2))))))))
(define (multi-merge . data)
    (foldl merge '() data))
```

2019-01-16

Define a pure function f with a variable number of arguments, that, when called like `(f x1 x2 .. xn)`, returns: `(xn (xn-1 ( .. (x1 (xn xn-1 .. x1))..)`. Function `f` must be defined using only fold operations for loops.

```
(define (f . L)
    (foldl (lambda (x y)
               (list x y))
           (foldl cons '() L)
           L))
```

Exercise session

2019-10-01

```
#lang racket

;; Let us define a variable.
(define hw "Hello world!")

;; Let us define a variable.
(define (hello)
  (display hw))

;; Try (display (hello))
;; Try (display hello)

;; We can do it as a lambda as well.
(define world
  (lambda () (display hw)))

;; Given an associative binary operator OP and a
    positive Natural number n, we want to
    compute 1 OP 2 OP 3 OP ... OP n
;; Let us try to do it with '+'.
(define (sum-range n)
  (if (<= n 1)
      1
      (+ n (sum-range (- n 1)))))

;; Now we generalize it
(define (fold-range op n e)
  (if (<= n 0)
      e
      (op n (fold-range op (- n 1) e))))

;; Try this:
(fold-range + 42 0)
(fold-range * 6 1)

;; Tail-recursive
(define (fold-range-tail op n e)
  (define (fold-range-tail-aux n acc)
    (if (<= n 0)
        acc
        (fold-range-tail-aux (- n 1) (op n acc))))
  (fold-range-tail-aux n e))

;; Try this:
(fold-range-tail + 42 0)
(fold-range-tail * 6 1)

;; "Iterative"
(define (fold-range-it op n e)
  (let count ((x n) (acc e))
    (if (<= x 0)
        acc
        (count (- x 1) (op x acc)))))

;; Try this:
(fold-range-it + 42 0)
(fold-range-it * 6 1)

;; Now try this:
(fold-range-it (lambda (x y)
                 (+ (* x x) y))
               10
               0)
(fold-range-tail (lambda (x y)
                   (+ (* x x) y))
                 10
                 0)
(fold-range (lambda (x y)
              (+ (* x x) y))
            10
            0)

;; Let us do something with lists
(define thelist '(1 2 3))
(cons 1 2)
(list? (cons 1 2))
(pair? thelist)
(cons 1 (cons 2 (cons 3 '())))
(car thelist)
(car (cdr (cdr thelist)))
(cadr thelist) ; aka (car (cdr l))
(caddr thelist) ; aka (car (cdr (cdr l)))

;; And now this:
```

```
(fold-range-it cons 42 '())
(fold-range-tail cons 42 '())
(fold-range cons 42 '())

;; Reverse pair
(define (riap p)
  (if (not (pair? p))
      (error (append (~a p) " is not a pair!"))
      (let ((f (car p))
            (s (cdr p)))
        (cons s f))))

;; Reverse list
(define (tsil l)
  (cond [(not (list? l)) (error (string-append
      (~a l) " is not a list!"))]
        [(null? l) l]
        [else (append (tsil (cdr l)) (list (car
            l)))]))

;; Flatten list
(define (flatten l)
  (cond [(null? l) l]
        [(not (list? l)) (list l)]
        [else (append (flatten (car l)) (flatten
            (cdr l)))]))

;; Try this:
(flatten '(1 2 3 4))
(flatten '((1) 2 ((3 4) 5) (((6)))))

;; Sorted list merge
(define (merge l1 l2)
  (cond [(not (and (list? l1) ( list? l2)))
      (error "Supplied with non-list argument!")]
        [(null? l1) l2]
        [(null? l2) l1]
        [else (let ((f1 (car l1)) (f2 (car l2)))
                (if (<= f1 f2)
                    (cons f1 (merge (cdr l1) l2))
                    (cons f2 (merge l1 (cdr
                        l2))))))]))

;; More general
(define (merge-gen compar l1 l2)
  (cond [(not (and (list? l1) ( list? l2)))
      (error "Supplied with non-list argument!")]
        [(null? l1) l2]
        [(null? l2) l1]
        [else (let ((f1 (car l1)) (f2 (car l2)))
                (if (compar f1 f2)
                    (cons f1 (merge-gen compar
                        (cdr l1) l2))
                    (cons f2 (merge-gen compar l1
                        (cdr l2)))))]))

(define (half-split l)
  (when (not (list? l)) (error "Supplied with
      non-list argument!"))
      (define (half-split-aux l1 l2 l3)
        (if (null? l3)
            (cons l1 l2)
            (half-split-aux (cons (car l3) l2) l1
                (cdr l3))))
      (half-split-aux '() '() l))

;; Merge-sort algorithm
(define (merge-sort compar l)
  (cond [(not (list? l)) (error (string-append
      (~a l) " is not a list."))]
        [(null? l) l]
        [(null? (cdr l)) l]
        [else (let ((halves (half-split l)))
                (merge-gen compar
                           (merge-sort compar (car
                               halves))
                           (merge-sort compar (cdr
                               halves))))]))
```

2019-10-08

```
#lang racket

;; Function that splits a list in two halves
(define (half-split-simple l)
  (let ((half (ceiling (/ (length l) 2))))
    (cons (take l half) (drop l half))))

;; Same, but hand-crafted.
(define (half-split l)
  (let split ((sec l)
              (last l))
    (cond [(null? last) (cons '() sec)]
          [(null? (cdr last)) (cons (list (car
              sec)) (cdr sec))]
          [else (let ((halves (split (cdr sec)
              (cddr last))))
                  (cons (cons (car sec) (car
                      halves)) (cdr halves)))])))

;; Structs

;; Binary tree

(struct node-base
  ((value #:mutable)))

(struct node node-base
  (left right))

(define (create-leaf v)
  (node-base v))

(define (create-internal v l r)
  (node v l r))

(define (leaf? n)
  (and (node-base? n) (not (node? n))))

(define (display-leaf l)
  (if (leaf? l)
      (begin
        (display "[Leaf ")
        (display (node-base-value l))
        (display "]"))
      (display "NOt a leaf!")))

(define (display-tree t)
  (cond [(leaf? t) (display-leaf t)]
        [(node? t) (begin
                     (display "[Node ")
                     (display (node-base-value t))
                     (display " ")
                      (display-tree (node-left t))
                     (display " ")
                     (display-tree (node-right t))
                     (display "]"))]
        [else (display "NOt a tree.")]))

(define foo (node 3 (node-base 2) (node-base 1)))

(define inc (lambda (x) (+ x 1)))

(define (tree-map f t)
  (if (leaf? t)
      (node-base (f (node-base-value t)))
```

```
     (node (f (node-base-value t))
           (tree-map f (node-left t))
           (tree-map f (node-right t)))))

(define (tree-map! f t)
  (begin
    (if (leaf? t)
        (set-node-base-value! t (f
            (node-base-value t)))
        (begin
          (set-node-base-value! t (f
              (node-base-value t)))
          (tree-map! f (node-left t))
          (tree-map! f (node-right t))))))

;; Macros

(define-syntax ++
  (syntax-rules ()
    ((_ i)
     (begin
       (set! i (+ 1 i))
       i))))

(define a1 1)
(++ a1)

(define (+++ x . rest)
  (begin
    (++ x)
    (if (null? rest)
        (list x)
        (cons x (apply +++ rest)))))

(define a2 2)
(define a3 3)
(+++ a1 a2 a3) ;; +++ this does not modify a1 a2
    and a3! We need a macro.

;; This way:
(define-syntax ++++
  (syntax-rules ()
    ((_ x)
     (begin
       (++ x)
       (list x)))
    ((_ x . rest)
     (begin
       (++ x)
       (cons x (++++ . rest))))))

(++++ a1 a2 a3)

;; Or this way:
(define-syntax ++++p
  (syntax-rules ()
    ((_)
     '())
    ((_ x r ...)
     (begin
       (++ x)
       (cons x (++++p r ...))))))

(++++p a1 a2 a3)

;; Pascal's repeat-until.
(define-syntax repeat
  (syntax-rules (until)
    ((_ stmt ... until cond)
     (let loop ()
       (begin
         stmt ...
         (unless cond
           (loop)))))))

(let ((x 1) (y 10))
  (repeat (++ x)
          (display x)
          (newline)
          until (> x y)))
```

2019-10-08

```
#lang racket

;; Let us see some continuations
(define (right-now)
  (call/cc
   (lambda (cc)
     (cc "Now"))))

(define (test-cc)
  (display (call/cc
            (lambda (escape)
              (display "cat\n")
              (escape "paw\n")
              "bacon\n"))))

;; We can make a generator of factorials
(define fact-gen #f)
(define (get-fact-gen)
  (let ((f 1) (n 1))
    (call/cc
     (lambda (cont)
       (set! fact-gen cont)))
    (set! f (* f n))
    (set! n (+ n 1))
    f))

;; Try (get-fact-gen) -> 1
;;     (fact-gen)     -> 2
;;     (define fact-gen2 fact-gen)
;;     (fact-gen)     -> 6
;;     (fact-gen2)    -> 24

;; This never terminates. Why?
(define (some-fact)
  (get-fact-gen)
  (displayln "asd")
  (fact-gen))

;; We can make a break statement
(define (break-negative)
  (call/cc
   (lambda (break)
     (for-each (lambda (x)
                 (if (>= x 0)
                     (begin
                       (display x)
                       (newline))
                     (break)))
               '(0 1 2 -3 4 5)))))

;; Or a continue statement
(define (skip-negative)
  (for-each (lambda (x)
              (call/cc
               (lambda (continue)
                 (if (>= x 0)
                     (displayln x)
                     (continue 'neg)))))
            '(0 1 2 -3 4 5)))

;; And a while with break and continue
(define-syntax while-do
  (syntax-rules (od break: continue:)
```

```
   ((_ guard od stmt ... break: br-stmt
       continue: cont-stmt)
    (call/cc
     (lambda (br-stmt)
       (let loop ((guard-val (guard)))
         (call/cc
          (lambda (cont-stmt)
            (when guard-val
              stmt ...)))
          (when guard-val
            (loop (guard)))))))))

(define (display-positive l)
  (let ((i l))
    (while-do (lambda ()
                (pair? i))
              od
              (let ((x (car i)))
                (unless (number? x)
                  (stop))
                (set! i (cdr i))
                (unless (>= x 0)
                  (cont))
                (displayln x))
              break: stop
              continue: cont)))
;; Try (display-positive '(0 1 2 3 -5 4 'asd 9))

;; Nondeterministic choices

; First, we need a FIFO queue
(define *paths* '())
(define (choose choices)
  (if (null? choices)
      (fail)
      (call/cc
       (lambda (cc)
         (set! *paths*
               (cons (lambda ()
                       (cc (choose (cdr choices))))
                     *paths*))
         (car choices)))))

(define fail #f)
(call/cc
 (lambda (cc)
   (set! fail
         (lambda ()
           (if (null? *paths*)
               (cc 'failure)
               (let ((p1 (car *paths*)))
                 (set! *paths* (cdr *paths*))
                 (p1)))))))


(define (is-the-sum-of sum)
  (unless (and (>= sum 0)
               (<= sum 10))
    (error "out of range"))
  (let ((x (choose '(0 1 2 3 4 5)))
        (y (choose '(0 1 2 3 4 5))))
    (if (= (+ x y) sum)
        (list x y)
        (fail))))


(define *queue* '())
(define (enqueue x)
  (set! *queue* (append *queue* (list x))))
(define (dequeue)
  (unless (empty-queue?)
    (let ((x (car *queue*)))
      (set! *queue* (cdr *queue*))
      x)))
(define (empty-queue?)
  (null? *queue*))

(define (start-coroutine proc)
  (call/cc
   (lambda (cc)
     (enqueue cc)
     (proc))))

(define (yield)
  (call/cc
   (lambda (cc)
     (enqueue cc)
     ((dequeue)))))

(define (exit-coroutine name)
  (displayln (string-append name ": exit."))
  (if (empty-queue?)
      (exit)
      ((dequeue))))

(define (say-something name n)
  (lambda ()
    (let loop ((i 0))
      (display name)
      (display ": ")
      (displayln i)
      (yield)
      (if (< i n)
          (loop (+ i 1))
          (exit-coroutine name)))))

(define (test-cr)
  (start-coroutine (say-something "A" 3))
  (start-coroutine (say-something "B" 2))
  (exit-coroutine "main"))

;; Try (test-cr)
```