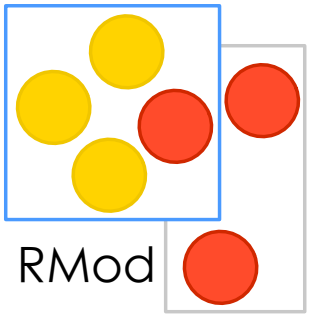
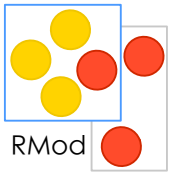
INRIA



An Introduction to Scheme

Stéphane Ducasse
stephane.ducasse@inria.fr
http://stephane.ducasse.free.fr/

Scheme



Minimal
Statically scoped
Functional
Imperative
Stack manipulation

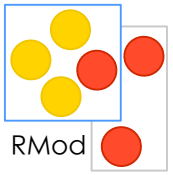
Specification in a couple of pages

http://schemers.org

RoadMap

- Execution principle
- Basic elements
- Pairs and Lists
- Function definition
- Some special forms
- One example of function

Conventions



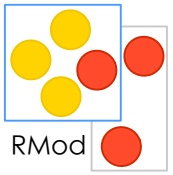
...? for predicates
equal?, boolean?

...! for side-effect
set!

global

char-, string-, vector- procedures

Read-Eval-Print



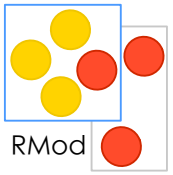
Read an expression
Evaluate it
Print the result and goto 1.

Using a Scheme interpreter

```
> (+ 1 3)
4
```

In this document

Implication



An interpreter always gets the ***value*** of an expression

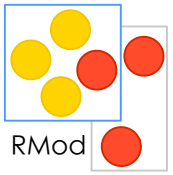
Different kinds of expression, *forms*, gets evaluated differently

- some are ***self-evaluating***: boolean, characters, numbers
- *user-defined* functions evaluate ***all*** their arguments
- *special* forms evaluate some of their arguments (if, when, set!, define, lambda, ...)

Prefix (+ 1 2)

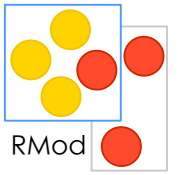
Value of first element applied on ***values*** of the rest

Examples




```
> 4                ;; self-evaluable
4
> -5
-5
> (* 5 6)              ;; applying *
30
> (+ 2 4 6 8)          ;; applying +
20
> (* 4 (* 5 6))        ;; nested expressiosn
120
> (* 7 (- 5 4) 8)
56
> (- 6 (/ 12 4) (* 2 (+ 5 6)))
```

Function Examples



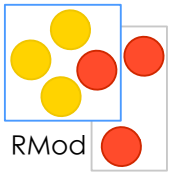

```
(define (square nb)
    ;; return the square of a number
    ;; nb -> nb
    (* nb nb))
```

```
(square 9)
=> 81
(square 4)
=> 16
```

RoadMap

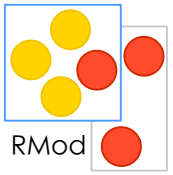
- Execution principle
- Basic elements
- Pairs and Lists
- Function definition
- Some special forms
- One example of function

Basic Elements



- Simple data types
  - booleans
  - numbers
  - characters
  - strings

Booleans



#t and #f

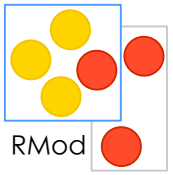
(not #f) => #t

(and (= 2 2) (> 2 1)) => #t

(or (= 2 2) (> 2 1)) => #t

(or (= 2 2) (< 2 1)) => #t

Booleans

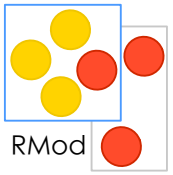


- #t and #f

- (boolean? #t) => #t
- (boolean? “hello”) => #f

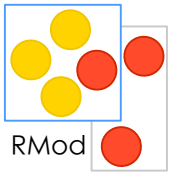
- self-evaluating
- > #t
- #t

Numbers



- self-evaluating
- 1.2 => 1.2
- 1/2 => 1/2
- 1 => 1
- 2+3i => 2+3i

Number Comparison



Numbers can tested for equality using the general-purpose equality predicate *eqv?*

(eqv? 42 42) => #t

(eqv? 42 #f) => #f

(eqv? 42 42.0) => #f

However, if you know that the arguments to be compared are numbers, the special number-equality predicate =is more apt.

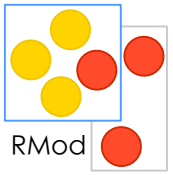
(= 42 42) => #t

(= 42 42.0) => #t

Other number comparisons allowed are

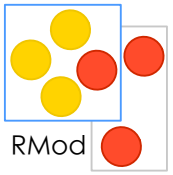
<,<= ,>,>= .

Numbers Operations



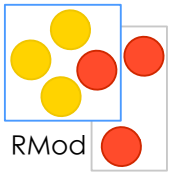
(+ 1 2 3) => 6
(- 5.3 2) => 3.3
(- 5 2 1) => 2
(* 1 2 3) => 6
(/ 6 3) => 2
(/22 7) => 22/7
(expt 23) => 8
(expt 4 1/2) => 2.0
(-4) => -4
(/4) => 1/4
(max 1 3 4 2 3) => 4
(min 1 3 4 2 3)=> 1
(abs 3) => 3

Number Predicates




```
(number? 42) => #t
(number? #t) => #f
(complex? 2+3i) => #t
(real? 2+3i) => #f
(real? 3.1416) => #t
(real? 22/7) => #t
(real? 42) => #t
(rational? 2+3i) => #f
(rational? 3.1416) => #t
(rational? 22/7) => #t
(integer? 22/7) => #f
(integer? 42) => #t
```

Strings



“abc d d”

self-evaluating

“abc” => “abc”

(string #\s #\q #\u #\e #\a #\k) => “squeak”

(string-ref “squeak” 3) => #\e

(string-append “sq” “ue” “ak”) => “squeak”

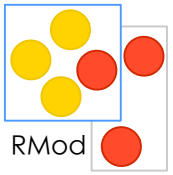
(define hello “hello”)

(string-set! hello 1 #\a)

hello => “hallo”

(string? hello) => #t

Symbols

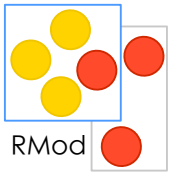


identifiers for variable
x, this-is-a-symbol ,i18n ,<=>
Not self-evaluating
Evaluation returns their value

Symbols are unique compared to strings

```
> (eq? (string->symbol "blabla")
       (string->symbol "blabla"))
#t
> (eq?  "blabla" "blabla")
#f
```

Assigning Value



(define symbol expr)

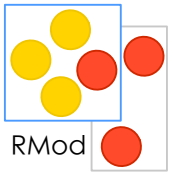
define a new symbol having as value the value of expr

(set! symbol expr)

change the value of symbol to be the one of expr

```
> (define xyz (+ 5 5)
> xyz
10
> (set! xyz (* 2 xyz))
> xyz
20
```

Quote



How to manipulate a variable and not its value?

Quote it!

‘a => a

‘(* 2 3) => (* 2 3) ;; a list

Remember: ‘Something => Something

‘Elephant => Elephant

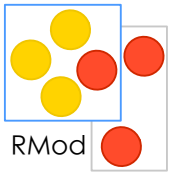
“Elephant => ‘Elephant

RoadMap

- Execution principle
- *Basic elements*
- ***Pairs and Lists***
- Function definition
- Some special forms
- One example of function

RMod

Pairs



Two values: an ordered couple, a pair

(cons 'a 'b) => (a . b)

(car (cons 'a 'b)) => a

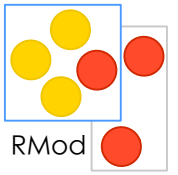
(cdr (cons 'a 'b)) => b

Not self-evaluating

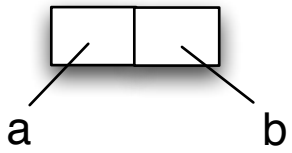
'(1 . #t) => (1 . #t)

(1 . #t) => Error

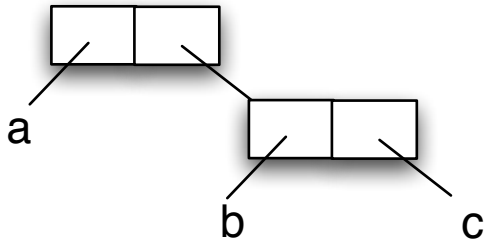
A Graphical Notation



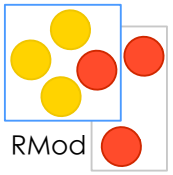
(cons 'a 'b) => (a . b)



(cons 'a (cons 'b 'c) => (a . (b . c))



Nested Pairs



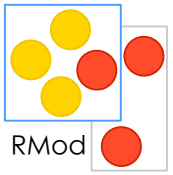
(cons (cons 1 2) 3) => ((1 . 2) . 3)
(car (cons (cons 1 2) 3)) => (1 . 2)
(cdr (car (cons (cons 1 2) 3))) => 2
(car (car (cons (cons 1 2) 3))) => 1
(caar (cons (cons 1 2) 3)) => 1

Notation

(cons 1 (cons 2 (cons 3 (cons 4 5))))

=> ***(1 2 3 4 . 5)*** <=> ***(1. (2. (3. (4 . 5))))***

The Empty List



Empty list: ()

‘() => ()

(null? ‘(a b)) => #f

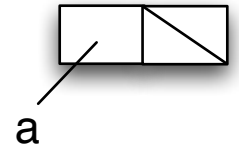
(null? ‘()) => #t

(pair? ‘()) => #f

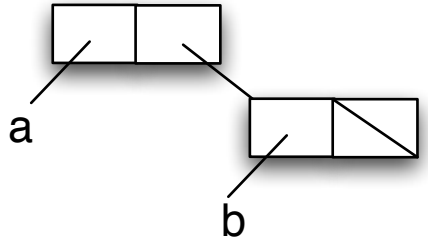
List and Pairs

A *list* is either
- the *empty list* or
- a pair whose *cdr* is a *list*

(a . ())) = (a)



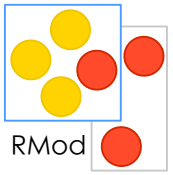
(cons 'a (cons 'b '()) => (a b)



(cons 1 '()) => (1 . ()) <=> (1)

(1 2 3 4)
<=> (1 . (2 . (3 . (4 . ()))))
<=> (cons 1 (cons 2 (cons 3 (cons 4 '()))))

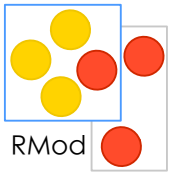
Consequences...



(car '(a b c d)) => a
(cdr '(a b c d)) => (b c d)
(cons 'a '(b c d)) => (a b c d)

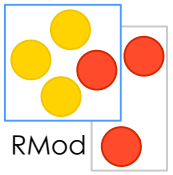
(car '()) => Error
(cdr '()) => Error
(pair? '()) => #f
(null? '()) => #t

List Predicates



(pair? '(1.2)) => #t
(pair? '(1 2)) => #t
(pair? '()) => #f
(list? '()) => #t
(null? '()) => #t
(list? '(1 2)) => #t
(list? '(1.2)) => #f
(null? '(1 2)) => #f
(null? '(1.2)) => #f

(List ...)

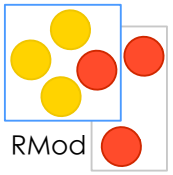


(list ...)

*evaluates **all** its arguments and create a list with results*

```
> (list 1 (+ 1 2) 3 4)
=> (1 3 3 4)
> (list ‘a ‘b ‘c)
=> (a b c)
> ‘(1 2 3 4)
=> (1 2 3 4)
```

(Append ...)



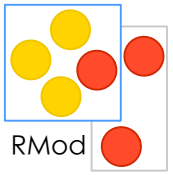
(append l1 l2 ..)

Evaluates its arguments and returns the concatenation of the results

(append ‘(a b c) ‘(d e) ‘(a))
> (a b c d e a)

(append ‘(a b c) ‘())
> (a b c)

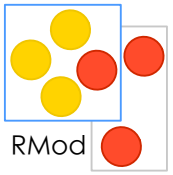
List Evaluation



- ***List evaluation is function application***

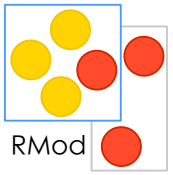
- Examine the first element of the s-expression.
- If the head ***evaluates*** to a procedure, the rest of the form is ***evaluated*** to get the procedure's argument values, and the procedure is applied to them.
- User-defined functions evaluate this way.
- Applicative order

List evaluation




```
(+ (* 3 2) 4)
+ => procedure
     (* 3 2)
      => same process => 6
                4
                => 4
                    => 10
```

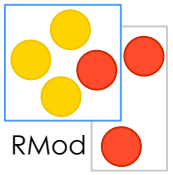
Quote and List



Quote forces lists to be treated as data

```
(define x ‘(cons ‘a ‘b))
x => (cons ‘a ‘b)
(list? x) => #t
(pair? x) => #t
(quote '()) => '()
''() => '()
```

Food for thought

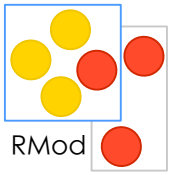


(cons ‘car ‘cdr)
(list ‘this ‘(is silly))
(cons ‘is ‘(this silly?))
(quote (+ 2 3))
(cons ‘+ ‘(2 3))
(quote cons)
(quote (quote cons))
((car (list + - *)) 2 3)

RoadMap

- Execution principle
- Basic elements
- Function definition
- Some special forms
- One example of function

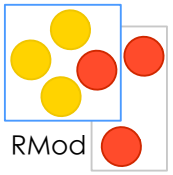
Procedure



cons => #<primitive:cons>
cons is a global variable referring to the primitive cons procedure

(car '(+ 1 2)) => +
(car (list + 1 2)) => #<primitive:+>

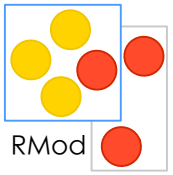
User Defined Procedure



(define (name arguments)
body)

```
(define (id x) x)
f => #<procedure:f>

> (id 3)
3
> (id ‘a)
a
```

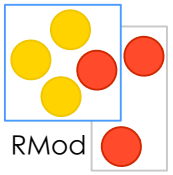
Main Definition Form




```
(define (name x y)
    body)
```

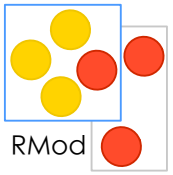
```
(define (square x)
    (* x x))
```

Recursive definitions...



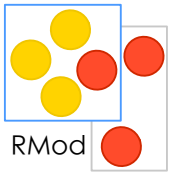

```
(define (length l)
   (if (null? l)
      0
      (+ 1 (length (cdr l)))

(define (useless l)
   (if (null? l)
      '()
      (cons (car l) (useless (cdr l)))
```

Towards map




```
(define (myMap f l)
    (if (null? l)
        '()
        (cons (f (car l)) (myMap f (cdr l)))))
```

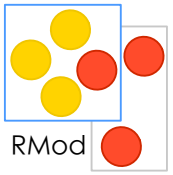
(map ...




```
(map abs '(1 -2 3))
> (1 2 3)
(map abs '(1 (-2) 3))
> error
(map cons '(1 2 3) '(a b c))
> ((1 . a) (2 . b) (3 . c))
```

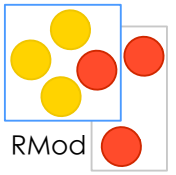
```
(map (lambda (x) (* x x)) '(1 2 3))
> (1 4 9)
```

define and lambda




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

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

Compose

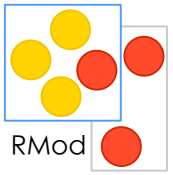



```
(define compose
    (lambda (g f)
        (lambda (x)
            (g (f x)))))


((compose (lambda (x) (+ x 2))
          (lambda (x) (* 3 x)))
7)
=> 23
```

RoadMap

- Variables
- Environment
- Let, Let*, Letrec
- The notion of closure
- High-order functions

define, lambda and let

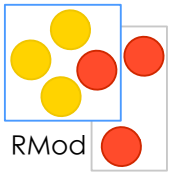



```
(define (f x)
      (define (f-help y)
        (/ (+ y 1) (- y 1)))
      (f-help (square x)))

(define (f x)
      ((lambda (y)
         (/ (+ y 1) (- y 1)))
       (square x)))
```

Binds y to (square x)

```
(define (f x)
      (let ((y (square x)))
         (/ (+ y 1) (- y 1))))
```

Lambda...



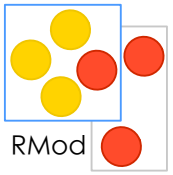
((lambda (formal parameters) body)

arguments)

creates bindings from formal parameter with arguments and evaluates body

```
(define x 20)      ;; modifies the lexical binding of a variable
(define add2
   (lambda (x)
      (set! x (+ x 2))
   x))

(add2 3) => 5
(add2 x) => 22
x => 20
```

Let & Lambda

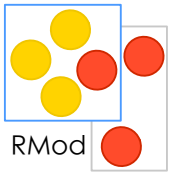


(let ((variable-1 expression-1)
(variable-2 expression-2)
. . .
(variable-n expression-n))
body)

is syntactic sugar for

((lambda (variable-1...variable-n) body)
expression-1
. . .
expression-n)

Let and Lambda

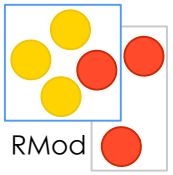



```
(let ((x (* 4 5))
      (y 3))
  (+ x y))
```

is equivalent to

```
((lambda (x y) (+ x y))
 (* 4 5) 3)
```

(Let ...



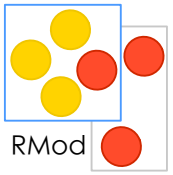

```
(let [(x 2)
      (y 3)]
  (* x y))                    =>  6
```

Bindings are evaluation order is not relevant.

```
(let [(x 2)
      (y 3)]
  (let [(x 7)
        (z (+ x y))]
    (* z x)))                 =>  35
```

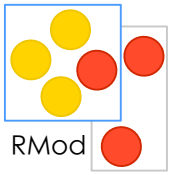
[is the same as (it is just more readable
When binding z only (x 2) is visible.

Let Examples



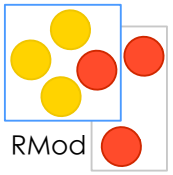

```
(let [(x 1)
      (y 2)
      (z 3)]
  (list x y z))        => (1 2 3)

(define x 20)
(let [(x 1)
      (y x)]
  (+ x y))             => 21

(let [(cons (lambda (x y) (+ x y)))]
  (cons 1 2))      => 3
```

(Let*



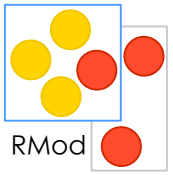

```
(let [(x 2)
      (y 3)]
  (let* [(x 7)
         (z (+ x y))]
    (* z x)))          =>  70

(let* [(x 1)
       (y x)]
  (+ x y))             => 2

equivalent to
(let [(x 1)]
  (let [(y x)]
    (+ x y)))          => 2
```

Where let is not enough




```
(let ((sum (lambda (ls)
             (if (null? ls)
                 0
                 (+ ( car ls) (sum (cdr ls)))))))
  (sum '(1 2 3 4 5 6)))
> reference to undefined identifier sum
```

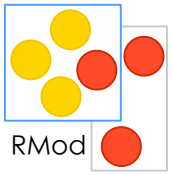
Passing itself




```
(let ((sum (lambda (s ls)
                (if (null? ls)
                    0
                    (+ ( car ls) (s s (cdr ls)))))))
  (sum sum '(1 2 3 4 5 6)))
```

Works but not really satisfactory

Let is not enough

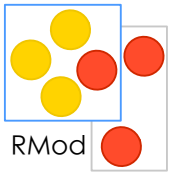



```
(let [(local-even? (lambda (n)
                     (if (= n 0) #t (local-odd? (- n1)))))
      (local-odd? (lambda (n)
                     (if (= n 0) #f (local-even? (- n 1))))))]
  (list (local-even? 23) (local-odd? 23)))
```

local-even? and local-odd? don't refer to the variables we are defining but other in parent environment.

Changing the let to a let* won't work either, local-even? inside local-odd? 's body refers to the correct procedure value, the local-odd? in local-even? 's body still points elsewhere.

(letrec ...

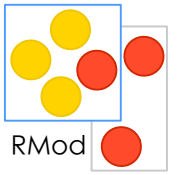


(letrec ((var expr) ...) body)

extends the current environment with vars with undefined values and associate to the vars the value of exprs evaluated within this environment.

```
(letrec [(local-even? (lambda (n)
                        (if (= n 0)
                            #t
                            (local-odd? (- n 1)))))
         (local-odd? (lambda (n)
                       (if (= n 0)
                           #f
                           (local-even? (- n 1))))))]
  (list (local-even? 23) (local-odd? 23)))
```

Variables and Env (i)




```
> n
ERROR:  Undefined global variable n
> (define n 5)
> n
5
> (* 4 n)
20
```

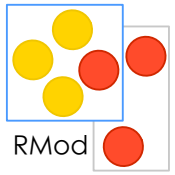
(define name expression)

(set! name expression)

(lambda...

(let...

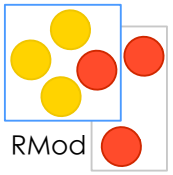
Variable and Env (ii)



In Scheme one single value for a symbol used as operator or operand

```
> ($ 4 5)
ERROR:  Undefined global variable $
> (define $ +)
> ($ 4 5)
9
> (define som +)
> (som 4 5)
9
```

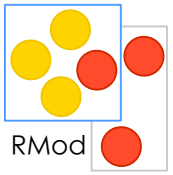
Lexical Scope




```
(define (test a b)
      (+ a b c))
> (test 1 2)
reference to undefined identifier: c

> (define (top-test c)
       (test 1 2))
> (top-test 3)
reference to undefined identifier: c
```

See more with closure

Closure



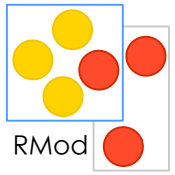
Closure: Environment + function

```
(let [(a 5)]
  (let [(f (lambda (x) (+ x a)))
        (a 0)]
    (f 10)))
=> 15
```

Lambda refers to it environment!

```
(let [(a 5)]
  (let [(f (lambda (x) (+ x a)))]
    (set! a 0)
    (f 10)))
=> 10
```

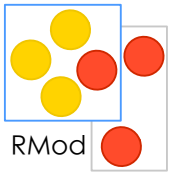
Creation Time Binding



The same bindings that were in effect when the procedure was created are in effect when it is applied

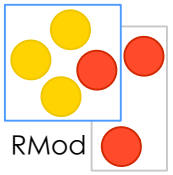
```
(let ((m  (let ((x 2))
            (lambda (y) (list x y)))))
 (let ((x 3))
   (m 4)))

> (2 4)
```

Spying Cons...




```
(define cons-count 0)
(define cons
  (let ([old-cons cons])
    (lambda (x y)
      (set! cons-count (+ cons-count 1))
      (old-cons x y))))
>(cons 'a 'b)
(a . b)
> cons-count
1
>(list 1 2 3)
> cons-count
1
```

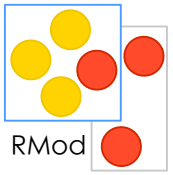
A Simple Counter




```
(define counter 0)
(define bump-counter
    (lambda ()
        (set! counter (+counter 1))
        counter))

(bump-counter) => 1
(bump-counter) => 2
```

Simple because everybody can change it and
we can only have one

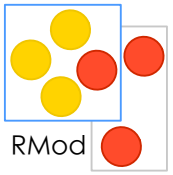
Counters...




```
(define next 0)
(define count
 (lambda ()
   (let ([v next])
     (set! next (+ next 1))
     v)))
next visible by others!

(define count
   (let ([next 0])
         (lambda ()
            (let ([v next])
              (set! next (+ next 1))
              v)))))
```

Now next is private to count

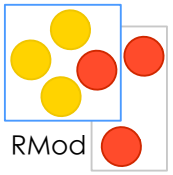
Counters...




```
(define make-counter
   (lambda ()
     (let ([next 0])
        (lambda ()
          (let ([v next])
                (set! next (+ next 1))
            v))))))
```

```
> (define c1 (make-counter)
> (c1) (c1)
1
> (define c2 (make-counter))
> (c2)
1
```

Kind of objects

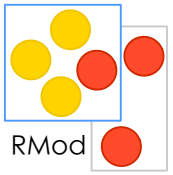



```
(define (make-fib-counter)
  (let ((nb 0))
    (letrec ((fib (lambda (n)
                    (set! nb (+ 1 nb))
                    (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))))
      (lambda message
        (case (car message)
          ((fib) (fib (cadr message)))
          ((nb) nb)
          ((reset) (set! nb 0)))))))

>(define fibc (make-fib-counter))
>(fibc 'fib 6)
8
> (fibc 'nb)
25  ;; number of recursive calls
```

RoadMap

- Execution principle
- Basic elements
- Function definition
- Some special forms
- One example of function

Special Forms



Some forms have specific evaluation strategy

Example

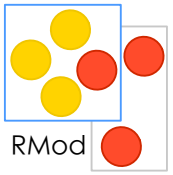
(**if** test truecase falsecase) does not evaluate all its arguments! Hopefully!

(lambda does not evaluate its argument)

Some special forms are: define, set!, cond, if, lambda....

We will see how to define special forms in the future

Sequence



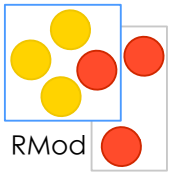
(begin exp1 ... expn)

evaluates exp1 ...expn and returns the value of expn

```
(begin
  (display "really!")
  (newline)
  (display "squeak is cool!"))
```

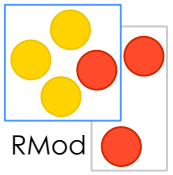
Scheme is not a pure functional language, it favors functional style but enables also imperative programming

(If...)




```
(if (char<? c#\c )
    -1
    (if (char=? c#\c )
        0
        1))
```

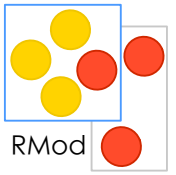
(Cond ...)




```
(cond
   ((predicate-expr1) actions1 ...)
   ((predicate-expr2) actions2 ...)
   ...
   (else actionsN...))
```

when predicate-expri is true, executes actionsi and returns the value of the last actionsi

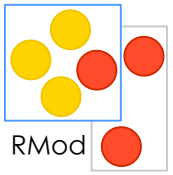
Cond Example




```
(cond
    ((char<? c#\c ) -1)
    ((char=? c#\c ) 0)
    (else 1))
```

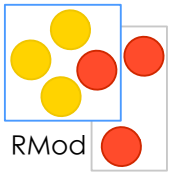
The cond actions are implicit **begin** s

(case ...)




```
(case #\e
  ((#\a) 1)
  ((#\b) 2)
  ((#\c #\e) 3)
  (else 4))
> 3

(case (car '(a b))
  ('a 1)
  ('b 2))
```

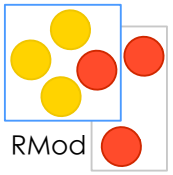
Let



(let ((variable-1 expression-1)
(variable-2 expression-2)
. . .
(variable-n expression-n))
body)

```
(let ((x (* 4 5))
      (y 3))
  (+ x y))
```

Macros

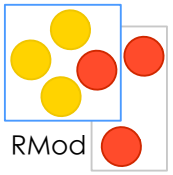


Mechanism needed when the language needs to be extended with new constructions

Provide an easy way to control evaluation of arguments.

New special forms can be defined as macros

Adding ifnot: **ifn**



Problem: How to add **ifn** such as:

(ifn expr then else) => then if expr is false, else else

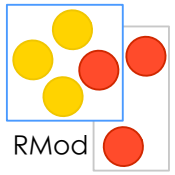
A first (not working) solution:

```
(define (ifn expr then else)
        (if (not expr)
            then
            else))
```

What does the following do?

(ifn (= 1 2) (display “foo”) (display “bar”))

Defining a special form



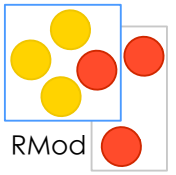
Arguments passed to **ifn** should ***not*** be evaluated. So **ifn** cannot be described with a simple function. A macro has to be used:

```
(define-macro (ifn test then else)
    (list 'if test else then)
```

Two phases:

1) macro-expansion: ***syntactic rewriting***
2) ***evaluation*** in the current environment of the resulting expression

ifn decomposed



The expression:

(ifn (= 1 2) (display “foo”) (display “bar”))

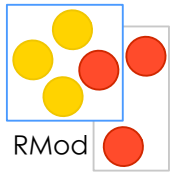
is *first* rewritten as:

(if (= 1 2) (display “bar”) (display “foo”))

and *then* evaluated:

>foo

In the need for macrocharacters



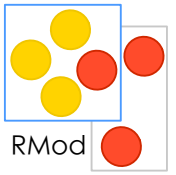
Here an example that can be simplified:

```
(define-macro (when test s1 . Lbody)
  (list 'if test
        (append (list 'begin s1) Lbody)
        #f))
```

Same with macrocharacters

```
(define-macro (when test s1 . Largs)
   `(if ,test
        (begin ,s1 ,@Larg)
        #f))
```

Macrocharacters ` , @



Useful for lighter macro definitions

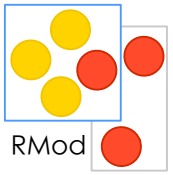
` is almost equivalent to ‘ except arguments preceded by , are evaluated.

```
(define-macro (ifn test then else)
    `(if ,test ,then ,else)
```

@ retrieve the contents of the list

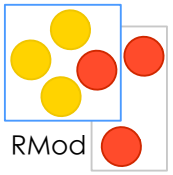
```
(let ((a ‘abc) (b ‘(1 2 3)) `(,a ,b ,@b))
=> (abc (1 2 3) 1 2 3)
```

New Iterators



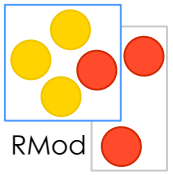

```
> (map list '(1 (2) 3))
((1) ((2)) (3))

(define (append-map f L)
    (apply append (map f L))
>(append-map list '(1 (2) 3))
(1 (2) 3)

(define (flatten s)
   (if (list? s)
       (append-map flatten s)
       (list s)))
>(flatten '(((1)) (1) 1))
(1 1 1)
```

New Iterators



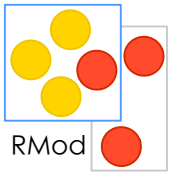

```
(define (map-select f L p?)
   (append-map (lambda (x)
                  (if (p? x)
                      (list (f x))
                      '()))
               L))
> (map-select (lambda (x) (+ 10 x)) '(1 2 3 4)
              (lambda (x) (odd? x)))
(11 13)

(define (every f L)
    (if (null? L)
        #t
        (and (f (car L)) (every f (cdr L)))))
```

Sums...



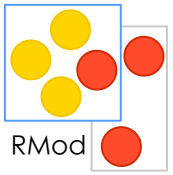

```
(define (sum-ints a b)
      (if (> a b) 0 (+ a (sum-ints (+ a 1) b))))
(sum-ints 3 7) => 25

(define (sum-sqrs a b)
      (if (> a b) 0 (+ (square a) (sum-sqrs (+ 1 a)  b))))
(sum-sqrs 3 7) => 135

(define (sum term next a b)
      (if (> a b)
          0
          (+ (term a) (sum term next (next a) b))))
(sum id plus-one 3 7)
```

Compose



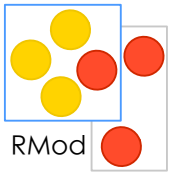

```
(define compose
    (lambda (g f)
        (lambda (x)
            (g (f x)))))


((compose (lambda (x) (+ x 2))
          (lambda (x) (* 3 x)))
7)
=> 23
```

Compose at Work




```
(define fth (compose car cddddr))

> (fth '(1 2 3 4 5))
5
```

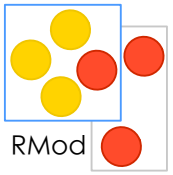
Derivation



Dg(x) = (g(x+dx) - g(x)) / dx

```
(define dx 0.00001)
(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
       dx)))

(define (cube x)
  (* x x x))
```

Capture

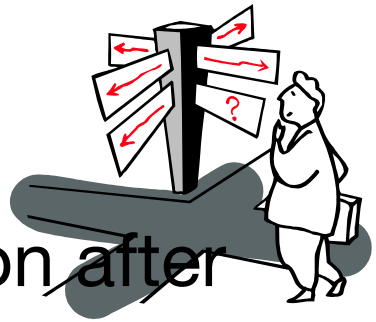



```
(define capture
  (lambda l
    (lambda (f)
      (f l))))
```

Currification

```
(define oneToSix (capture 1 2 3 4 5 6))
>(oneToSix car)
1
>(oneTosix (lambda (x)
```

Continuations



- How to escape the current computation after an error?
- How to discard pending computation?
- How to control the future computation?

Applications

Advanced control flow

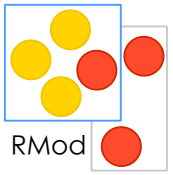
Escaping mechanism

Web server

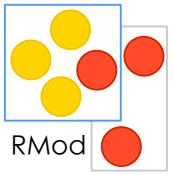
Difficulty escaping a computation

```
> (+ ( 1000 (abort (* 2 3))) (+ 2000 3000))
> 6
```

Continuations

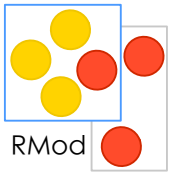


Two Approaches



- Explicit continuation of a computation
  - Continuation Passing Style CSP
    - by adding a new parameter that represents the current continuation of the computation
- Using call/cc which captures the current continuation

CSP: Manual Continuation



(define (foo x)
is transformed into
(define (k-foo x cont) so that
k-foo returns (cont (foo x))

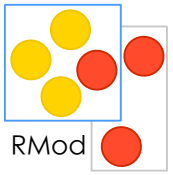
Pros

- control the future of the computation
- can give up and choose another future
- can be stored of passed to another functions

Cons

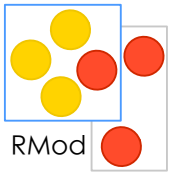
- tedious

Factorial-k




```
(define (fac n)
      (if (= 0 n)
          1
          (* n (fac (- n 1)))))
(define (kfac n k) ...
  ??
(kfac 5 (lambda (x) x))
120
(kfac 5 (lambda (x) (+ 2 x)))
122
```

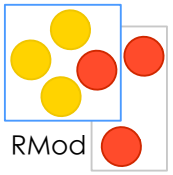
Getting Factorial-k



(kfac 0 k) = (k (fac 0)) = (k 1)
(kfac n k) = (k (fac n))
= (k (* n (fac (- n 1)))
= (kfac (- n 1) (lambda (v) (k (* n v))))

```
(define (kfac n k)
;; n * k = function of one arg
(if (= 0 n)
    (k 1)
    (kfac (- n 1) (lambda (v) (k (* n v))))))
```

How computation works..



(+ 3 (* 4 (+ 5 6)))

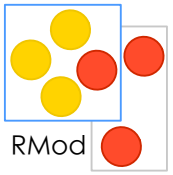
first + 5 6 results in 11

then * 4

then + 3

Once (+ 5 6) is evaluated the *rest* of the computation to continue is * 4 and + 3...

Contexts



A context is a procedure of one variable [] (hole)

How to create a context?

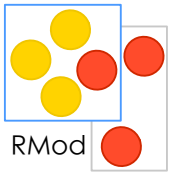
Two steps:

- replace expression by []
- form a lambda

(+ 3 (* 4 (+ 5 6)))

context of (+ 5 6) = (lambda ([]) (+ 3 (* 4 [])))

Contexts (ii)



Refined two Steps

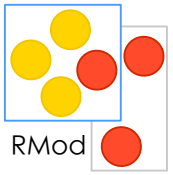
- replace expression by [] and evaluate it as much as we can
- then we form a lambda

```
(if (zero? 5)
    (+ 3 (* 4 (+ 5 6)))
    (* (+ (* 3 4) 5) 2))))
```

Context of (* 3 4) is

```
(lambda ([]) (* (+ [] 5) 2))
```

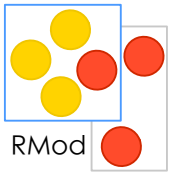
Context of (* 3 4)




```
(let ((n 1))
   (if (zero? n)
       (writeln (+ 3 (* 4 (+ 5 6))))
       (writeln (* (+ (* 3 4) 5) 2)))
  n)
is


(lambda ([])
  (begin
    (writeln (* (+ [] 5) 2)
    n)
```

with the closure environment having n bound to 1

Execute ...

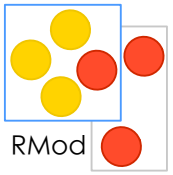



```
(define (mapadd l)
  (if (null? l)
      (cons (+ 3 (* 4 5)) '())
      (cons (+ 1 (car l)) (mapadd (cdr l)))))

(mapadd '(1 3 5))
> (2 4 6 23)

Context of (* 4 5) in (cons 0 (mapadd ‘(1 3 5)))?

(lambda  ([])
     (cons 0 (cons 2 (cons 4 (cons 6 (cons (+ 3 []) ‘()))))))
```

Roadmap

- Context
- Escaper
- Continuation

Escape Procedures



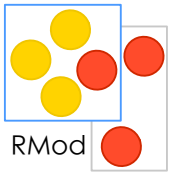
Yields a value but never passes that value to others.

The value of an escape procedures is the value of the *entire* computation

Anything awaiting its result is ignored

```
(+ (escape-* 5 2) 3)
> 10

(* 100 (+ 30 (escape-* 5 2)))
> 10
```

Escaper

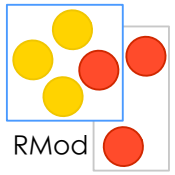



```
(+ ((escaper
        (lambda (x)
            ((escaper -) ((escaper *) x 3)
                         7)))
             5)
      4)
> 15
```

Roadmap

- Context
- Escaper
- Continuation

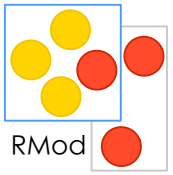
Continuation



(.....(call/cc (lambda (k) ....k....)...)

We pass the rest of the computation, i.e., the *context* of (call/cc..) as an *escaper* to k

Continuation Evaluation



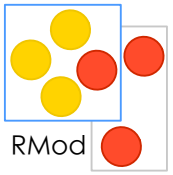
(1) form the context of (call/cc cont)
⇒ (lambda []...)

(2) create an escaper
⇒ (escaper (lambda [] ...)

(3) pass (escaper (lambda [] ...) to cont
where cont = (lambda (k)...)

If cont ***uses*** the escaper k then it escapes!
Else the normal value mechanism is applied

Example



(+ 3 (* 4 (call/cc (lambda (k) ....)))

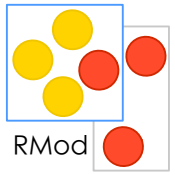
Context of (call/cc ...)
⇒ (lambda ([]) (+ 3 (* 4 [])))

We have then

```
(+3 (* 4 ((lambda (k) ....)
            (escaper (lambda ([]) (+ 3 (* 4 [])))))
```

The context of call/cc is escaped and passed as k

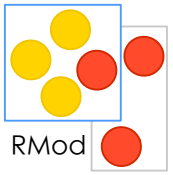
Call/cc



(call-with-current-continuation proc)
Proc must be a procedure of ***one*** argument. The procedure call-with-current-continuation packages up the current continuation as an "***escape procedure***" and passes ***it as an argument*** to proc. The escape procedure is a Scheme procedure that, if it is later called, will ***abandon*** whatever continuation is in effect at that later time and will instead use the continuation that was in effect when the escape procedure was created. [...]

The escape procedure that is passed to proc has unlimited extent just like any other procedure in Scheme. It may be stored in variables or data structures and may be called as many times as desired.

Example

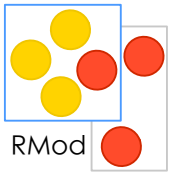


- (call/cc (lambda (k) (+ 2 (* 7 (k 3)))))
- > 3
- k = (escaper (lambda ([]) []))

- (+ 1 (call/cc (lambda (k) (+ 2 3))))
- > 6
- k = (escaper (lambda ([]) (+ 1 [])))
- but k not invoked

- (+ 1 (call/cc (lambda (k) (+ 2 (**k** 3)))))
- > 4
- k = (escaper (lambda ([]) (+ 1 []))) = (lambda^ ([]) (+ 1 []))
- k invoked

Escaping on Error




```
(define (product l)
  (call/cc (lambda (exit)
          (letrec ((proaux (lambda (l)
                        (cond ((null? l) 1)
                              ((not (number? (car l)))
                               (exit 'error))
                              ((zero? (car l)) (exit 0))
                              (else (* (car l) (proaux (cdr l))))))))
            (proaux l)))))

(product '(42 27 q 42 12 0 9 10 11 22))
> error
(+ 3 (product '(42 27 q 42 12 0 9 10 11 22)))
>+: expects type <number> as 2nd argument, given: erreur;
other arguments were: 3
```