COMPUTER SCIENCE 51

Spring 2009

cs51.seas.harvard.edu

Prof. Greg Morrisett
Prof. Ramin Zabih

Notes

Midterm
- good job!
- will go over in sections
- see Profs if you struggled

This week:
- mutation & side effects
  - see VII of *HTDP*, 4.9 & 5.8 of *PLT Guide*
- key concepts of OO programming
  - see 5.3 & 13 of *PLT Guide*

Side Effects

So far, we've been working with the *purely functional* subset of Scheme.

- *principle of substitution*: you can always* replace an expression `e` with its value (and vice versa) without affecting the output of a program.
  - sometimes known as *referential transparency.*
- results in a very simple evaluation model
  - to evaluate $(e_1\ e_2\ \ldots\ e_n)$:
  - evaluate e_i to a value v_i .
  - `e1` should evaluate to a $(\lambda\ (x_2\ \ldots\ x_n)\ e)$
  - *substitute* v_i for x_i in `e`.
  - evaluate the resulting expression.

* well, almost always...

Substitution in Action

Consider these two sets of definitions:

```
(define x (factorial 4))
(define y (factorial 4))

(define x 24)
(define y x)
```

Are these equivalent?

That is, can I write a function that tells which set of definitions I used for `x` & `y`?

No! A good thing...

From a compiler's standpoint, the substitution principle is *great.*

It can replace computations over constants with their value (constant folding)

- `(fact 4) ==> 24`

It can factor out common sub-expressions:

```
(* (length x) (length x))
      ==>
(let ([n (length x)])
     (* n n))
```

Side Effects

A *side effect* breaks the substitution principle.

For example, diverging computations are a side effect:

```
(define x (car (cons 3 (f))))
```

Is *not* equivalent to: `(define x 3)` because `(f)` could run forever.

Sliding Scale of Effects

- expressions that diverge or throw an exception:
  - Because the expression may not have a value!
  - But substitution holds when they do have values.
  - So these are more "benign" than other effects.
- "local" variable assignment
  - Breaks substitution -- variable has different values at different times.
  - But doesn't suffer from *sharing* (alias) problems.
  - Can almost always eliminate.
- shared data structure mutation, input & output
  - Non-local, subtle effects due to sharing
  - Bad interactions with multi-threading
  - But to be fair, these are crucial facilities!

Local Variable Assignment

Two new forms:

`(set! x e)`

Modifies the variable `x` to have the value of `e`, and returns `#<void>`.

`(begin` e_1 e_2 `...` e_n`)`

Execute e_1, then e_2, then ... then e_n, and return the value of e_n.

set! Example

```
(define x 0)
    x ==> 0
(set! x (+ 1 x))
    x ==> 1

(define (incx)
  (begin (set! x (+ 1 x)) x)))
   (incx) ==> 2
   (incx) ==> 3
   (+ (incx) (incx)) ==> ?
```

Consider:

```
(define (counter n)
  (λ () (begin (set! n (+ 1 n)) n)))

(define c1 (counter 0))
(define c2 (counter 0))
(c1) ==> 1
(c1) ==> 2
(c2) ==> 1
(c2) ==> 2
(+ (c1) (c2)) ==> 6
```

Encapsulation

- The general pattern:
  ```
  (let ([x init])
     (lambda (...)
         ...(set! x ...)...))
  ```
  gives us a variable `x` that is *local* to a function.
- Encapsulating state gives us more control:
  - cannot access `x` except by calling the function.
  - if we must maintain an invariant on `x`, we don't have to worry about the invariant being violated by an outsider.

Another Example

```
(define stack empty)

(define (push v)
  (set! stack (cons v stack)))

(define (pop)
  (let ([v (car stack)])
    (begin (set! stack (cdr stack))
           v)))
```

Better

```
(define-struct stack (push pop))

(define (new-stack)
  (let*
    ([s empty]
     [push (λ (v) (set! s (cons v s)))]
     [pop (λ () (let ([v (car s)])
                  (begin (set! s (cdr s))
                         v)))])
      (make-stack push pop)))
```

Objects as Structs + λ

These definitions effectively define an object: some encapsulated state (s) and methods (push & pop) that operate over the state.

```
(define-struct stack (push pop))

(define (new-stack)
  (let*
    ([s empty])
     [push (λ (v) (set! s (cons v s)))]
     [pop (λ () (let ([v (car s)])
                  (begin (set! s (cdr s))
                         v)))])
      (make-stack push pop)))
```

Corresponding Java

```
class List {
  public final Object car;
  public final List cdr;
}

class Stack {
  List s = null;
  public void push(Object v) {
     s = new List(v,s);
  }
  public Object pop() {
      Object v = s.car;
      s = s.cdr;
     return v
  }
}
```

Avoid Gratuitous set!s

I will personally come to your dorm and string you up by your thumbnails if you start coding like this:

```
(define n 0)

(define (for stop f)
  (if (<= n f)
      (begin (f n)
             (set! n (+ 1 n)))
      42))
```

Keep it Functional if Possible

```
(define (for n stop f)
  (if (<= n f)
      (begin (f n)
             (for (+ 1 n) stop f))
      42))
```

Note that even Java, C, & Fortran compilers compile down to functional intermediate forms where possible so they can do optimizations like constant folding and common sub-expression elimination!

Rules of Thumb:

- Stick to functional programming!
  - no gratuitous uses of set!
  - your code will be easier to reason about.
  - and the compiler will be happier.
- If you must introduce set! then:
  - avoid global variables
    - must remember to "re-initialize"
    - hard to keep clients from modifying them
    - leads to hell in a multi-threaded world
  - encapsulate uses as local variables
    - c.f., the stack example
- Guidelines hold regardless of the language.

Shared Mutable Data

As bad as shared, mutable variables are, the issues pale in comparison to shared, mutable data structures.

All the same problems as global variables:
- break substitution
- re-initialization
- not thread-safe

But compounded by *aliasing.*
- multiple names (i.e., paths) for an object.
- disruption along any of these paths wreaks havoc.
- can't tell when two paths lead to the same object.
- But, of course, crucial for many data structures!

(An Aside)

- You can often predict where languages are heading by looking at what compilers and architectures want.
  - compilers *love* pure functional code
    - enables serious optimizations (e.g., fusion)
  - compilers *love* statically typed code
    - avoids indirection and run-time checks
  - architectures *hate* memory
    - it's way slower than the CPU
    - difficult to make fast & shared
  - architectures *love* locality
    - it makes dealing with memory tolerable

Mutable Lists

PLT provides mutable lists:
- `mcons, mcar, mcdr`
- `set-mcar!, set-mcdr!`

but they are not the default.

In most languages, mutable lists are the default.
- so most people can do "in-place" updates.
- sometimes this is needed for efficiency's sake.
- but as we'll see, this often results in wasted space.
- real care must be taken...

Example: A Queue

```
(define front empty)
(define rear empty)

(define (enqueue x)
  (if (empty? rear)
    (begin (set! front (mcons x empty))
           (set! rear front))
    (begin (set-mcdr! rear (mcons x empty))
           (set! rear (mcdr rear)))))

(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0
           x)))
```

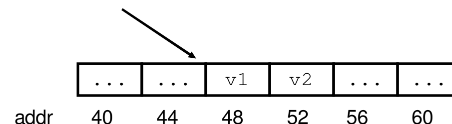
What's Going On?

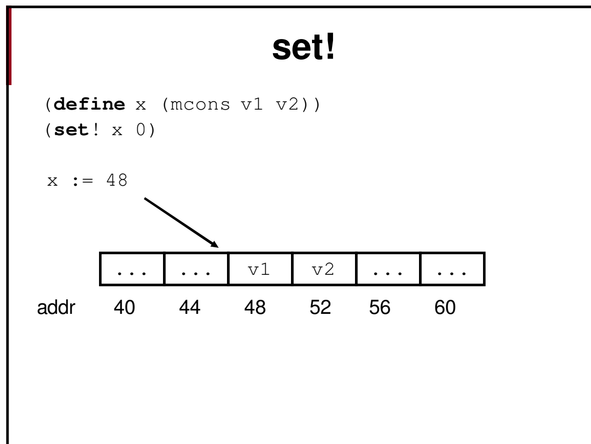
- `(mcons v1 v2)`
  - allocate space in memory for two words at some address
  - store the value `v1` in the first word
  - store the value `v2` in the second word
  - return the adress (i.e., pointer to the pair)
- In C, we would write:

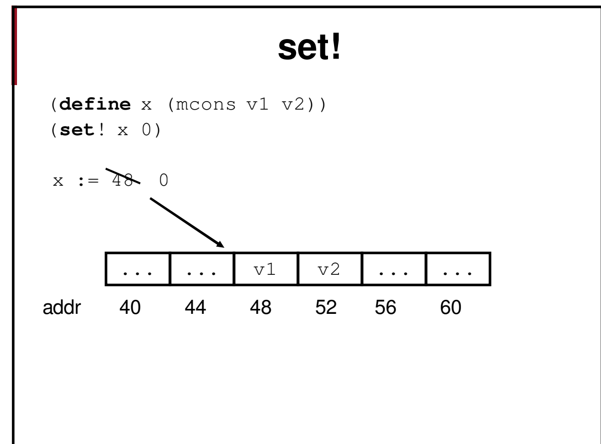
```
struct mpair { void *car; void *cdr; };
struct mpair* mcons(void *v1, void *v2) {
  struct mpair* res = malloc(sizeof(struct mpair));
  mpair->car = v1;
  mpair->cdr = v2;
  return res;
}
```

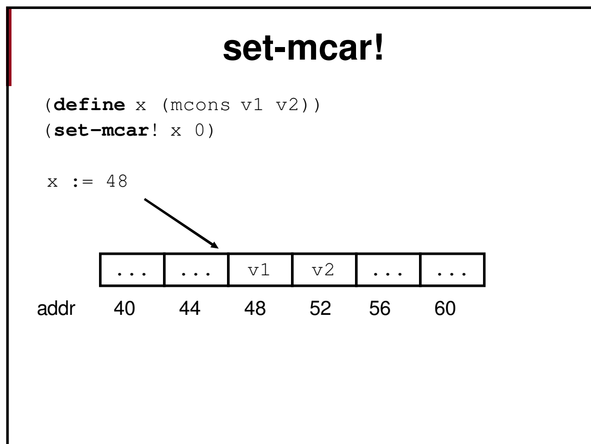
In Pictures

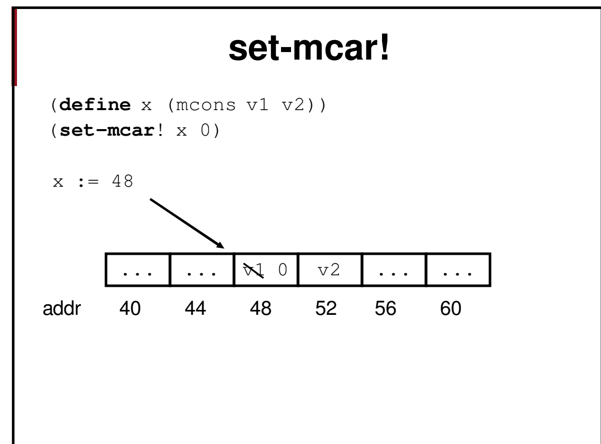
```
(define x (mcons v1 v2))
```

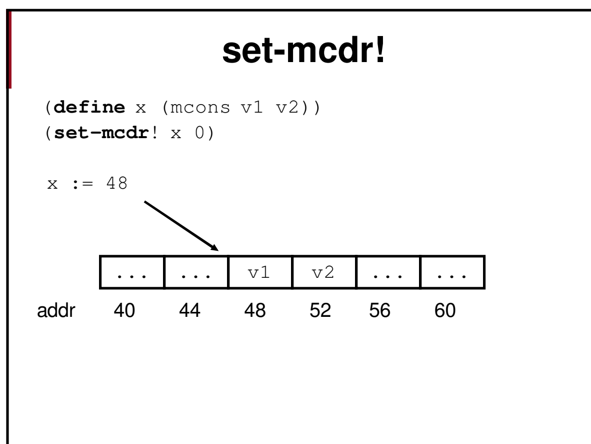


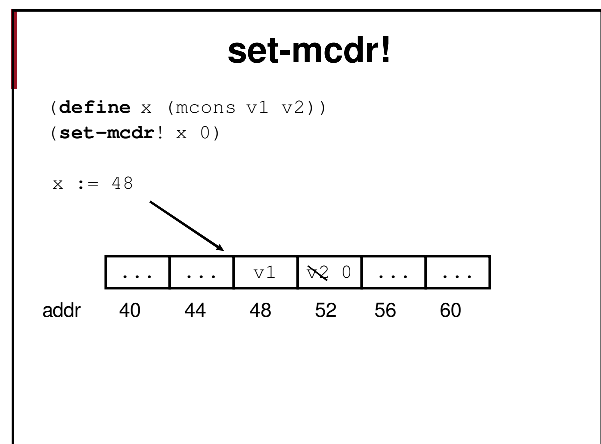

set!
(define x (mcons v1 v2))
(set! x 0)
x := 48
... ... v1 v2 ... ...
addr 40 44 48 52 56 60


set!
(define x (mcons v1 v2))
(set! x 0)
x := 48 0
... ... v1 v2 ... ...
addr 40 44 48 52 56 60


set-mcar!
(define x (mcons v1 v2))
(set-mcar! x 0)
x := 48
... ... v1 v2 ... ...
addr 40 44 48 52 56 60


set-mcar!
(define x (mcons v1 v2))
(set-mcar! x 0)
x := 48
... ... v1 0 v2 ... ...
addr 40 44 48 52 56 60


set-mcdr!
(define x (mcons v1 v2))
(set-mcdr! x 0)
x := 48
... ... v1 v2 ... ...
addr 40 44 48 52 56 60


set-mcdr!
(define x (mcons v1 v2))
(set-mcdr! x 0)
x := 48
... ... v1 v2 0 ... ...
addr 40 44 48 52 56 60

enqueue

```
(define front empty)
(define rear empty)

(define (enqueue x)
  (if (empty? rear)
    (begin (set! front (mcons x empty))
           (set! rear front))
    (begin (set-mcdr! rear (mcons x empty))
           (set! rear (mcdr rear)))

(enqueue 3)
```

rear := emp
front := emp

enqueue

```
(define (enqueue x)
  (if (empty? rear)
    (begin (set! front (mcons x empty))
           (set! rear front))
    (begin (set-mcdr! rear (mcons x empty))
           (set! rear (mcdr rear)))

(enqueue 3)
```

rear := empty
front := 68

...	...	3	emp	...	...	...
60	64	68	72	76	80	84

addr

enqueue

```
(define (enqueue x)
  (if (empty? rear)
    (begin (set! front (mcons x empty))
           (set! rear front))
    (begin (set-mcdr! rear (mcons x empty))
           (set! rear (mcdr rear)))

(enqueue 3)
```

rear := 68
front := 68

...	...	3	emp	...	...	...
60	64	68	72	76	80	84

addr

enqueue

```
(define (enqueue x)
  (if (empty? rear)
    (begin (set! front (mcons x empty))
           (set! rear front))
    (begin (set-mcdr! rear (mcons x empty))
           (set! rear (mcdr rear)))

(enqueue 3)
(enqueue 4)
```

rear := 68
front := 68

...	...	3	emp	...	...	...
60	64	68	72	76	80	84

addr

enqueue

```
(define (enqueue x)
  (if (empty? rear)
    (begin (set! front (mcons x empty))
           (set! rear front))
    (begin (set-mcdr! rear (mcons x empty))
           (set! rear (mcdr rear)))

(enqueue 3)
(enqueue 4)
```

rear := 68
front := 68

...	...	3	emp	...	...	...
60	64	68	72	76	80	84

addr

enqueue

```
(define (enqueue x)
  (if (empty? rear)
    (begin (set! front (mcons x empty))
           (set! rear front))
    (begin (set-mcdr! rear (mcons x empty))
           (set! rear (mcdr rear)))

(enqueue 3)
(enqueue 4)
```

rear := 68
front := 68

...	...	3	80	...	4	emp
60	64	68	72	76	80	84

addr

enqueue

```
(define (enqueue x)
  (if (empty? rear)
    (begin (set! front (mcons x empty))
           (set! rear front))
   (begin (set-mcdr! rear (mcons x empty))
          (set! rear (mcdr rear)))))

(enqueue 3)
(enqueue 4)
```

rear := 68
front := 68

...	...	3	80	...	4	emp

addr 60 64 68 72 76 80 84

enqueue

```
(define (enqueue x)
  (if (empty? rear)
    (begin (set! front (mcons x empty))
           (set! rear front))
   (begin (set-mcdr! rear (mcons x empty))
          (set! rear (mcdr rear)))))

(enqueue 3)
(enqueue 4)
```

rear := 80
front := 68

...	...	3	80	...	4	emp

addr 60 64 68 72 76 80 84

dequeue

```
(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0)
           x)))

(dequeue)
```

rear := 80
front := 68

...	...	3	80	...	4	emp

addr 60 64 68 72 76 80 84

dequeue

```
(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0)
           x)))

(dequeue)
```

x := 3
rear := 80
front := 68

...	...	3	80	...	4	emp

addr 60 64 68 72 76 80 84

dequeue

```
(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0)
           x)))

(dequeue)
```

x := 3
rear := 80
front := 68

...	...	3	80	...	4	emp

addr 60 64 68 72 76 80 84

dequeue

```
(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0)
           x)))

(dequeue)
```

x := 3
rear := 80
front := 80

...	...	3	80	...	4	emp

addr 60 64 68 72 76 80 84

dequeue

```
(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0)
           x)))

(dequeue)
```


dequeue

```
(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0)
           x)))

(dequeue)
```


dequeue

```
(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0)
           x)))

(dequeue)
(dequeue)
```


dequeue

```
(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0)
           x)))

(dequeue)
(dequeue)
```


dequeue

```
(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0)
           x)))

(dequeue)
(dequeue)
```


dequeue

```
(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0)
           x)))

(dequeue)
(dequeue)
```

dequeue

```
(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0)
           x)))

(dequeue)
(dequeue)
```

x := 4
rear := 80
front := emp

...	...	3	80	...	4	emp
60	64	68	72	76	80	84

addr

dequeue

```
(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0)
           x)))

(dequeue)
(dequeue)
```

x := 4
rear := emp
front := emp

...	...	3	80	...	4	emp
60	64	68	72	76	80	84

addr

dequeue

```
(define (dequeue)
  (let ([x (mcar front)])
    (begin (set! front (mcdr front))
           (if (empty? front) (set! rear empty) 0)
           x)))

(dequeue)
(dequeue)
```

x := 4
rear := emp
front := emp

...	...	3	80	...	4	emp
60	64	68	72	76	80	84

addr

Invariants

- When the queue is empty, both front and rear are empty.
- When the queue is non-empty:
  - front points to an mcons
  - rear points to the last mcons in the list starting with front.
  - it's easy to mess this up!
- Of course, the right thing is to encapsulate front and rear so that these invariants can't be broken by the outside.

Encapsulated Queue

```
(define-struct queue (enqueue dequeue))

(define (new-queue)
  (let* ([front empty]
         [rear  empty]
         [enq (lambda (x)
                (if (empty? rear)
                    (begin (set! front (mcons x empty))
                           (set! rear front))
                    (begin (set-mcdr! rear (mcons x empty))
                           (set! rear (mcdr rear)))))]
         [deq (lambda ()
               (let ([x (mcar front)])
                 (begin (set! front (mcdr front))
                        (if (empty? front)
                            (begin (set! rear empty) x)
                            x))))])
    (make-queue enq deq)))
```

Some Trickiness

Consider the mlength function:

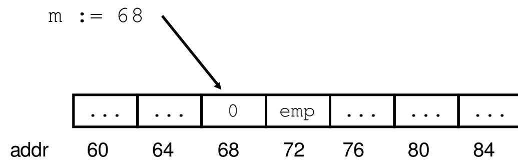
```
(define (mlength mx)
  (if (empty? mx) 0
      (+ 1 (mlength (mcdr mx)))))
```

And consider this sequence:

```
(define m (mcons 0 empty))
(set-mcdr! m m)
(mlength m)
```

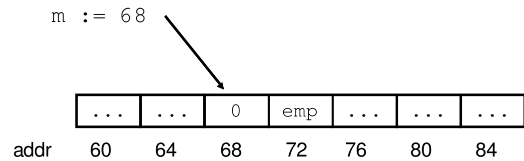
Cycles:

```
(define m (mcons 0 empty))
(set-mcdr! m m)
(mlength m)
```



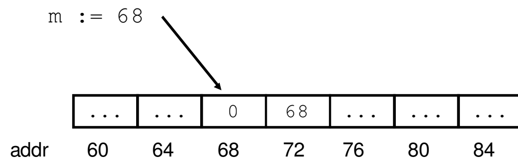
Cycles:

```
(define m (mcons 0 empty))
(set-mcdr! m m)
(mlength m)
```



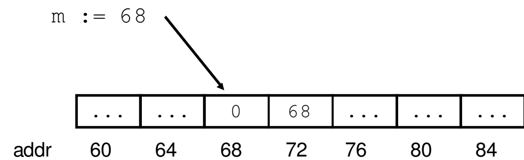
Cycles:

```
(define m (mcons 0 empty))
(set-mcdr! m m)
(mlength m)
```



Cycles:

```
(define m (mcons 0 empty))
(set-mcdr! m m)
(mlength m)  -- OOPS!!!
```



Another Gotcha:

Consider an “efficient” mappend:

```
(define (mappend! xs ys)
  (cond ([(empty? xs) ys]
         [(empty? (mcdr xs)) (set-mcdr! xs ys)]
         [else (mappend! (mcdr xs) ys)])))


(define xs (mcons 1 (mcons 2 empty)))
(define ys (mcons 3 (mcons 4 empty)))
(mappend! xs ys)
xs ==> {1 2 3 4}
(mappend! ys (mcons 5 (mcons 6 empty)))
ys ==> {3 4 5 6}
xs ==> {1 2 3 4 5 6}
```

Worse yet...

```
(define (mappend! xs ys)
  (cond ([(empty? xs) ys]
         [(empty? (mcdr xs)) (set-mcdr! xs
  ys)]
         [else (mappend! (mcdr xs) ys)])))

(define xs (mcons 1 (mcons 2 empty)))

(define ys (mappend! xs xs))
```

Morals

- In-place update is often more efficient.
  - e.g., constant real-time dequeue.
  - as opposed to our functional queues which were only amortized constant time.
- But coding is *much* harder:
  - must formulate & pay attention to invariants
  - need to worry about cycles
    - how to compare two mlists?
  - need to worry about accidental sharing
    - defensive approach: copy
    - but this ends up with *less* sharing
  - and this is just in the non-parallel world!
    - research papers on efficient parallel queues.

Typical Exam Problems

- Fill in ? to make the expression evaluate to 42:

```
(define f ?)
(if (= (f) 42) 0 (f))
```

- Write a function to flatten an acyclic mlist without ever calling mcons.

```
(mcons (mcons 1 (mcons 2 empty))
   (mcons (mcons 3 empty)
     (mcons 4 empty)))  ==>
(mcons 1
  (mcons 2
    (mcons 3 (mcons 4 empty))))
```