# Computer Programs Et09-12


dr. H. Corporaal
dr.ir. A.J.C. van Gemund
Computer Systems Group

http://cs.et.tudelft.nl/~heco
email: heco@cs.et.tudelft.nl


**Session 6: Objects and State**



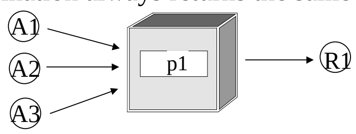
## Overview

- Assignment and local state change
- New primitives:
  - set!
  - set-car!
  - set-cdr!
- Consequences:
  - true object-oriented programming possible
  - procedures may have side effects
  - exit substitution model
    - procedure outcome becomes history dependent
  - contents of structured data may change: mutable data



## Until now

- A combination always returns the same value

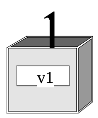



arguments procedure result

- The value of a variable never changes

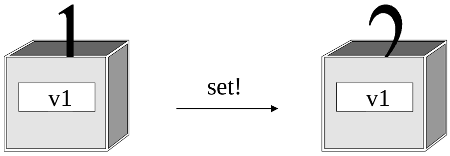
(apart from redefinition)

( define v1 1 )



# Introducing set!

( define v1 1 )
( set! v1 2 )




- Convention: all procedures that use assignment have names that end with ‘!’

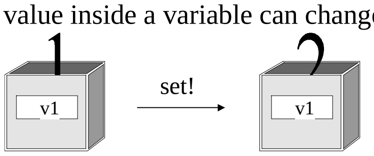


# Set! changes everything

- The result of evaluation a combination need not remain constant

( define v1 1 )
( + v1 v1 ) → 2
( set! v1 2 )
( + v1 v1 ) → 4

- The value inside a variable can change






# Set! changes everything

- Compare define and set!

```
(define v 0)

(define (change1)
  (define v 1)
  v)

(define (change2)
  (set! v 2)
  v)

(define (change3 v)
  (set! v 3)
  v)
```

**What is value of v after using these functions?**

## Sequence, step by step

```
( sequence                ; PC-scheme also uses 'begin'
    ( +   1  1 )
    ( +   2  2 )
    ( +   3  3 )
) ——→  6
```

- Sequence only makes sense if assignment is used (or other side-effects)

```
( define   v1  1 )
( define   v2  2 )
( sequence
    ( set!   v1  3 )
    ( +      v1  v2 )
) ——→  5
```

Slide 1.6. 7

## Objects change state !

```
(define counter 0)

(define (object)
  (set! counter (+ 1 counter))
  counter)

(object) -> 1
(object) -> 2
```

- Not a real object; does not contain **local** state

```
(set! counter 5)
(object) -> 6
```

Slide 1.6. 8

## Objects change state !

```
(define object
  (let ((counter 0))
    (lambda ()
      (set! counter (+ 1 counter))
      counter) ))
(object) -> 1
(object) -> 2
```

- A real object; does contain **local** state !!

```
(set! counter 5)   ; probably gives error message !
(object) -> 3
```

Slide 1.6. 9

# Objects change state !

- If many similar objects needed, use an object generator (creates objects)

```
(define (object-create)
  (let ((counter 0))
    (lambda ()
      (set! counter (+ 1 counter))
      counter) ))

(define x-object (object-create))
(define y-object (object-create))

(x-object) -> 1
(x-object) -> 2
(y-object) -> 1
```

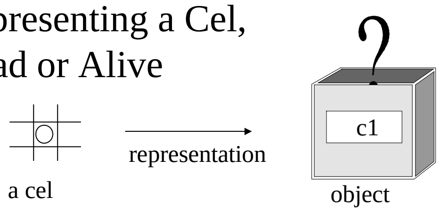


**Mr. Skiem, session 1.6.a**

**- Define and Set!**
**- Building objects with state**



# Representing a Cel, Dead or Alive




- System to be modeled is represented by object(s)
- Changes in system are represented by changes in the state of objects
- The state of an object is changed by assignment

## Game of Life: Representing a cell

```
( define    ( new-cell )
    ( let    (( life #F ))
        ( define    ( isalife? )
            life )
        ( define    ( isdead! )
            ( set!  life  #F ))
        ( define    ( isalife! )
            ( set!  life  #T ))
        ( define    ( dispatch  command )
            (cond
                (( eq  command  'isalife? )  isalife? )
                (( eq  command  'isalife! )  isalife! )
                (( eq  command  'isdead! )  isdead! )))
        dispatch ))
```

Slide 1.6. 13

## Using object representation cell

( define  c1  ( new-cell ))

- c1 is an object which represents a cell

  (( c1  'isalife? )) → #F

- The current state of c1 indicates if the cell is alife

  (( c1  'isalife! ))
  (( c1  'isalife? )) → #T

- The state of object c1 can change over time

Slide 1.6. 14

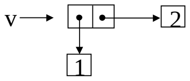
**Mr. Skiem, session 1.6.c**

**Game of Life**

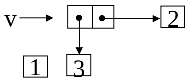
Slide 1.6. 15

# Even lists change

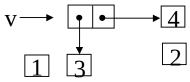
( define v ( cons 1 2 ))




( set-car! v 3 )



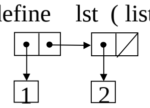

( set-cdr! v 4 )



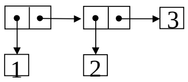

Slide 1.6. 16

# Lists, lists, lists

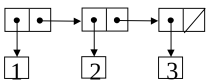
( define lst ( list 1 2 ))




( set-cdr! ( cdr lst ) 3 )



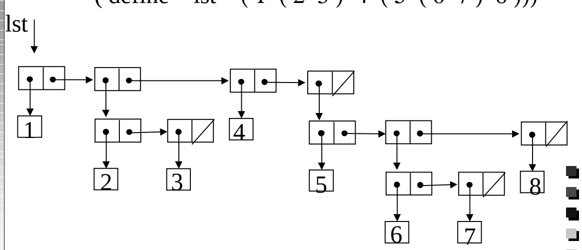

( set-cdr! ( cdr lst ) ′( 3 ))




Slide 1.6. 17

# More changing lists

( define lst ′( 1 ( 2 3 ) 4 ( 5 ( 6 7 ) 8 )))



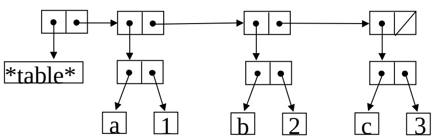

Slide 1.6. 18

## Modeling tables

- A table

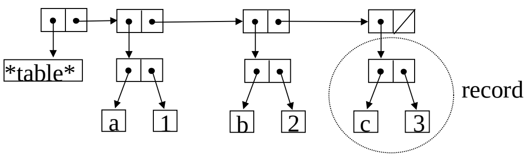
| key | a | b | c |
|---|---|---|---|
| value | 1 | 2 | 3 |

- A representation






## Implementing table, 1

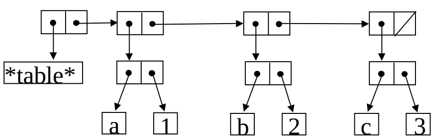



- Given a key, how to find its value?

```
( define    ( lookup   key  table )
    ( let    (( record   ( assq   key  ( cdr  table ))))
        ( if    ( null?   record)
                nil
                ( cdr   record))))
```



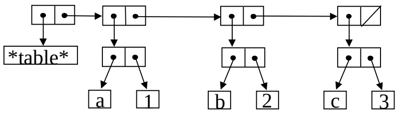
## Implementing table, 2




- Implementing assq   (= associative search on key)

```
( define  ( assq    key  records )
    ( cond   (( null?   records )  nil )
             (( eq?   key  ( caar  records )) ( car records ))
             ( else   ( assq   key  ( cdr  records )))  ))
```

# Implementing table, 3




- Inserting a new record:

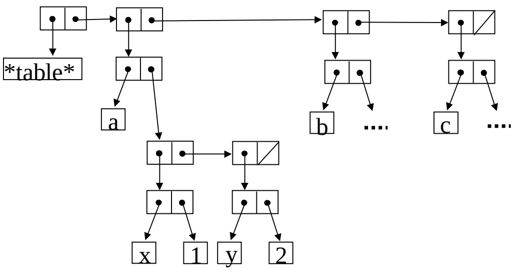
```
( define    ( insert!  key  value  table )
      ( let   (( record  ( assq  key  ( cdr  table ))))
           ( if    ( null?  record )
              ( set-cdr!   table
                             ( cons  ( cons  key  value ) ( cdr  table )))
              ( set-cdr!   record  value )))
      'OK )

( define ( make-table )          ; creates empty table
      ( list  '*table* ))
```



# Two dimension tables

- Replace the value of a 1-dim table with a sublist




( lookup 'a 'y table ) $\rightarrow$ 2



**Mr. Skiem, session 1.6.b**

**Modifying lists**

## Nothing stays the same

- Before assignment:

  ( define v1 ( strange-function 10 ))

  ( define v2 ( strange-function 10 ))

- Question: Are v1 and v2 the same ?
  - Answer: yes, they are both assigned the same value
  - Answer: yes, by using replacing, you can proof sameness

Slide 1.6. 25

## Nothing stays the same

```
( define    counter  0 )
( define    ( strange-function   x )
     ( sequence
           ( set!    counter  ( +   1  counter ))
           ( +    counter  x ) ))

( define    v1  ( strange-function  10 ))
( define    v2  ( strange-function  10 ))
```

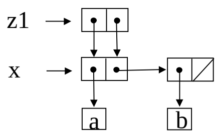
- v1 and v2 are definitely not the same

Slide 1.6. 26

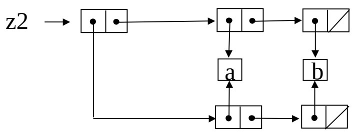
## Lists are also changing

- z1 and z2 are the same list ((a b) a b)

```
( define    x    ( list    'a 'b ))
( define    z1   ( cons    x  x ))
```

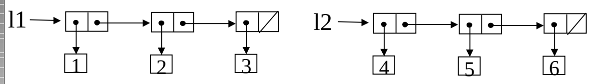



```
( define    z2  ( cons    ( list  'a 'b ) ( list  'a 'b )))
```




Slide 1.6. 27

## Watch out, changing lists




```
( define   l1   ( list  1  2  3 ))
( define   l2   ( list  4  5  6 ))

( set-cdr!   ( cdr  ( cdr  l2 ))  l1 )
( set-car!   ( cdr  ( cdr  ( cdr  l2 )))  7 )
```

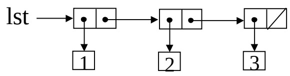
l1 → ( 7 2 3 )
l2 → ( 4 5 6 7 2 3 )

Slide 1.6. 28

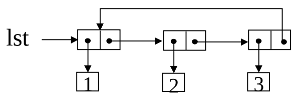
## Watch out, circular, infinite lists

- Changing lists may cause circular lists

```
(define lst (list  1 2 3))
```




```
(set-cdr! (cddr lst) lst)
```




Slide 1.6. 29

## What is eq?

- **eq?** compares two pointers
- Pointer to equal atomic data items (numbers, symbols, strings) are always the same
- Atomic data items can never be changed

- What is **equal?**   See exercise 2.54, pg 145
- What is the difference with the '=' operator?

Slide 1.6. 30

## Concl.: Exit substitution model

- The substitution model was based on replacing names by their value
- After introducing assignment, the value connected to a name can change. Global replacement is not valid anymore
- A replacement for the substitution model is needed

Slide 1.6. 31