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 4: Data Abstraction I

Slide 1.4. 1

Introduction

- So far we looked at procedure abstraction
  - name bound to algorithm
  - usage
    - manipulation of data
    - creation / usage of other procedures
- abstraction: black box approach, only input/output behavior is known
  - hide implementation details

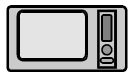
Slide 1.4. 2

Introduction

- Now: data abstraction
  - name bound to complex data structure
  - usage: build larger, more complex, real world data structures (compound data)
- Abstraction:
  - hide implementation details

Slide 1.4. 3

Compound data



Simple numeral data:

```
( define   p1         0 )    ; pixel value
( define   p2         1 )
    |
( define   p1000000   0 )
```

Compound data:

```
( define   screen   ’( 0 1 0 0 1 … 0 ))
```

Slide 1.4. 4

Compound procedure/compound data

- Compound procedure
  ex: (sum f 1 10)
- Compound data:
  ex: (DisplayOnTV screen)
- Principles for both:
  - work with complex objects
  - hide implementation details

Slide 1.4. 5

Structured data objects

- In C we have so-called ‘struct’
  - e.g.

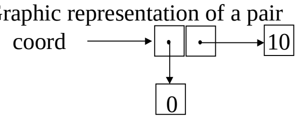
```
struct {
  int seq_nr;
  char data[1024];
} packet;
```

- In Lisp/Scheme we can do everything with a pair
  - a pair can contain everything (dynamic typing)

Slide 1.4. 6

Pair, basic building block

- Example: coordinate (define x 0)
  (define y 10)
- Cons, glueing together two data items
  (define coord (cons x y))
- Car, unglueing the head data item
  (car coord) → 0
- Cdr, unglueing the tail data item
  (cdr coord) → 10
- Graphic representation of a pair



Slide 1.4. 7

Pair, basic building block

- constructor: cons
- accessors: car & cdr
- representation:
  - graphic box
  - printed representation: (x . y)

- Generally it holds that:
  (car (cons x y)) = x
  (cdr (cons x y)) = y

Slide 1.4. 8

Using compound data

- Example old: distance between two points

```
( define    ( distance  x1  y1  x2  y2)
     ( sqrt  ( +   ( square   ( -   x2  x1 ))
                   ( square   ( -   y2  y1 )))))

( distance  0  10  1  15 )
```

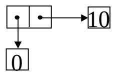
- Example new:

```
( define    ( distance  p1  p2 )
     ( sqrt  ( +   ( square   ( -   ( car  p2 ) ( car  p1 ))
                   ( square   ( -   ( cdr  p2 ) ( cdr  p1 )))))))
( define    p1   ( cons  0  10 ))
( define    p2   ( cons  1  15 ))
( distance  p1  p2 )
```

Slide 1.4. 9

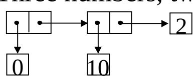
Glueing glued data items

- Two numbers, one pair



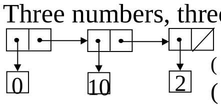
(cons 0 10)

- Three numbers, two pair minimum



(cons 0 (cons 10 2))

- Three numbers, three pair normal



(cons 0 (cons 10 (cons 2 nil)))
(list 0 10 2)

Slide 1.4. 10

Compound data, lists

- The list is the most important data form in scheme / lisp
- Lists are formed using <cons> of lists
- Lists are manipulated using <car> or <cdr>
- Lists are <nil> terminated

  (<nil> is sometimes called the empty list)

Slide 1.4. 11

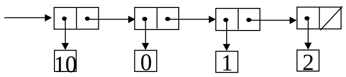
Mr. Skiem, session 1.4.a

Examples of Compound Data

Slide 1.4. 12

Length, manipulating lists

Given: (define m (list 10 0 1 2))

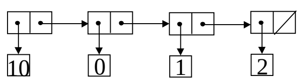
m 

Question: What is the length of <m> ?

Procedure, recursive:

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

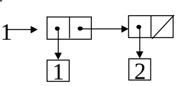

Length, manipulating lists

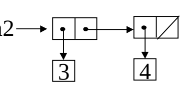
Procedure, iterative: 


```
( define     ( length   lst )
     ( define     ( length-iter   cnt  m )
          ( if      ( null?  m )
                    cnt
                    ( length-iter    ( +   1  cnt )
                                     ( cdr   m ))))
     (length-iter 0 lst))
```

- <null?> checks for empty list

Append, creating new lists

m1 

m2 


```
( define   m1   ( cons   1   ( cons   2  nil )))
( define   m2   ( list   3  4 ))
```

Recursive procedure:

```
( define     ( append   l1  l2 )
     ( if       ( null?   l1 )
                l2
                ( cons   ( car   l1)
                         ( append    ( cdr   l1 )  l2 ))))
```

MapCar, higher-order functions and lists

- Apply a procedure <f> to each element of <lst>

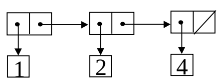
```
( define    ( mapcar   f  lst )
     ( if     ( null?   lst )
              nil
              ( cons     ( f     ( car   lst ))
                         ( mapcar   f    ( cdr    lst )))))

( mapcar     ( lambda     ( x ) ( +    1   x ))
             ( list    1  2  3  4  5 ))
```

Slide 1.4. 16

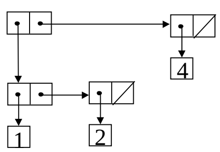
More glue, lists in lists

Simple list:



(list   1 2 4)

List within list:



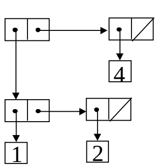
(list   (list   1 2) 4)

Slide 1.4. 17

Atomic, friend or foe ?

(atom?   x) →   #F   if x is a pair
                      →   #T   otherwise

```
( define    ( countatoms    x )
     ( cond    (( null?     x )  0 )
               (( atom?   x )  1 )
               ( else    ( +    ( countatoms    ( car x ))
                                ( countatoms    ( cdr x ))))))
```



(countatoms (list (list 1 2) 4))
↓
3

Slide 1.4. 18

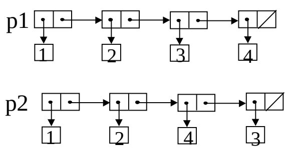
Pincode: using lists

- Needed:
  - representation for pincodes
  - comparing pincodes
- Representation:
  - a pincode is represented as a list of 4 numbers constructor:

```
( define    ( pincode   d1  d2  d3  d4 )
     ( list    d1  d2  d3  d4 ))
```

Slide 1.4. 19

Pincode: operations




```
( define  p1 ( pincode 1 2 3 4 ))
( define  p2 ( pincode 1 2 4 3 ))

( pincode-compare  p1 p2 ) →??
```

- Compare pincodes

```
( define   ( pincode-compare    p1  p2 )
     ( if   ( null?    p1 )
            #T
            ( if   ( =     ( car   p1 ) ( car  p2 ))
                   ( pincode-compare   ( cdr  p1 ) ( cdr  p2 ))
                   #F )))
```

Slide 1.4. 20

Mr. Skiem, session 1.4

- 1.4b. Lists
- 1.4c. Lists of Lists

Slide 1.4. 21