



CS1 Recitation 5

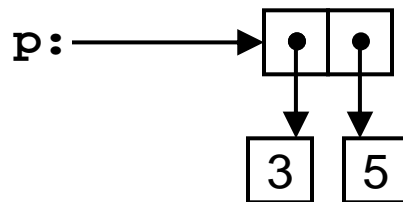
Fall 2008

cons Pairs

- Code:

```
(define p (cons 3 5))
```

- Diagram:



- Arrows point to *entire* cons cell
- Can't have arrows that point to only first or second part of cons cell
- Displayed as (3 . 5) in Scheme interpreter

cons and Lists

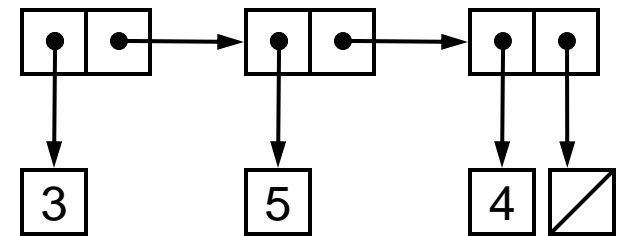
- Can use `cons` to create lists of values

```
(cons 3 (cons 5 (cons 4 (list))))
```

- `(list)` makes an empty list

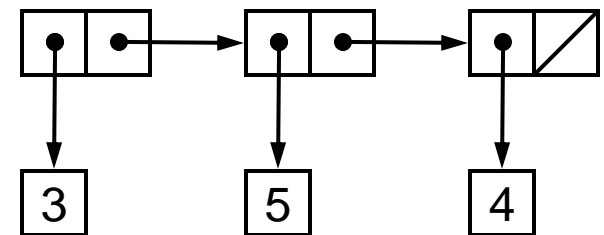
- Book uses `nil` instead

- `(define nil (list))`



- A simplification:

- Don't draw empty-list value as a separate box



Some Pitfalls

- What about this?

- `(cons 3 (cons 5 4))`

- This isn't a normal list

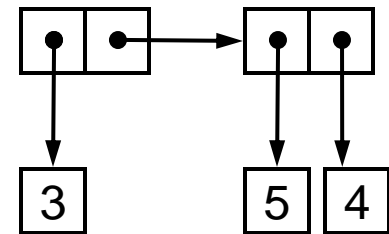
- Last value is a dotted-pair

- Displayed as `(3 5 . 4)`

- What does this do: `(car (list))`

- Error! Can only use `car` and `cdr` on `cons`-pairs.

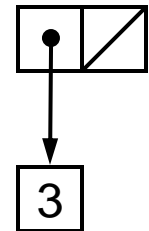
- An empty list is a specific value, not a `cons`-pair.



list and null?

- Much easier to use `list` to make lists
 - `(list)` makes an empty list
 - `(list 3 5 4)`
 - Same as `(cons 3 (cons 5 (cons 4 (list))))`
- `null?` returns `#t` if argument is an empty list
 - `(null? (list))`
→ `#t`
 - `(null? (cdr (list 3)))`
→ `#t`

`(list 3)`



List Processing

- Design process doesn't change!
 - Just new tools:
 - `cons` to put a new value onto the front of an existing list
 - `car` to get first element from a list
 - `cdr` to get the rest of the list
 - `null?` to see if a list is empty
- Process:
 - Identify and solve base case(s)
 - Identify recursive step
 - How to solve problem in terms of itself?

List Processing (2)

- More specific details for list processing
- Base cases are often (but not always!)...
 - “Is the list empty?”
 - “Am I supposed to do something with only the first element, in particular?”
- Recursive step usually involves:
 - Process the first value in the list...
 - ...and concatenate with result of calling myself on the rest of the list

Example: Sum a List

- Write a procedure to sum the values in a list.
- Base case?
 - Empty list: Result is 0
- Recursive case?
 - Add head of list to sum of remainder of list.
- Code:

```
(define (sum-list vals)
  (if (null? vals)
      0
      (+ (car vals)
         (sum-list (cdr vals)))))
```

Sum a List (2)

- Code:

```
(define (sum-list vals)
  (if (null? vals)
      0
      (+ (car vals) (sum-list (cdr vals)))))
```

- What kind of process?

- Linear recursive: deferred + operations

- How to turn it into a linear iterative process?

- Keep a running sum

- Need a state value → need a helper function

Iterative List-Sum

- Write the skeleton first:

```
(define (isum-list vals)
  (define (isum-helper v sum)
    ... )
  (isum-helper vals 0))
```

- Often very useful to figure out how the helper-function will be called, before actually trying to write it
 - Sanity-check to make sure the idea is good. 😊

Iterative List-Sum (2)

- Fill in skeleton, following linear recursive code:

```
(define (isum-list vals)
  (define (isum-helper v sum)
    (if (null? v)
        sum
        (isum-helper (cdr v)
                      (+ sum (car v)))))
  (isum-helper vals 0))
```

- No more deferred operations now.
- Moral: list processing functions can also be linear iterative or linear recursive

Example: Filter a List of Values

- Write a procedure that takes a list and a maximum value:
 - “Remove” all elements whose value is larger than the maximum
 - Return a new list containing only the values less than the max
- Base cases?
 - Empty list: Result is an empty list
- Recursive case?
 - If current value is greater than max, return the result of filtering the rest of the list
 - Otherwise, return a new list with the current value, plus the result of filtering the rest of the list

Filter a List of Values (2)

- Code:

```
(define (filter-list vals max-val)
  (cond ((null? vals) vals)
        ((> (car vals) max-val)
         (filter-list (cdr vals) max-val))
        (else
         (cons (car vals)
               (filter-list (cdr vals) max-val)))))
```

- Can we improve on this?

- Filtering a list is a pretty useful operation in general...
- Could create a generic filtering operation that takes a predicate
- Define this operation in terms of the generic filter procedure

Example: Map a Function to a List

- Write a procedure that takes a function and a list:
 - Generate a new list containing the result of applying the function to each input-list value
 - Example:

```
(map (lambda (x) (* x x)) (list 1 2 3 4))  
→ (1 4 9 16)
```
- Base case?
 - If list is empty, as usual...
- Recursive case is pretty simple, too
 - Apply function to head of list
 - Concatenate result with result of recursive call to `map`

Map a Function to a List (2)

- Code:

```
(define (map f inputs)
  (if (null? inputs)
      (list)
      (cons (f (car inputs))
            (map f (cdr inputs)))))
```

- So useful that Scheme has a built-in `map` function!

- Scheme version is more generic than this
- Scheme `map` inputs are a function of n inputs, and n lists of values
- Example:

```
(map (lambda (x y) (* x y))
     (list 1 4 9 5) (list 2 1 6 8)) → (2 4 54 40)
```

Subsets

- Book problem 2.32

```
(define (subsets s)
  (if (null? s)
      (list nil)
      (let ((rest (subsets (cdr s))))
        (append rest (map <??> rest)))))
```

- Takes a list of values

- e.g. (1 2 3)

- Returns all subsets of the input

- e.g. ((1 2 3) (1 2) (1 3) (2 3) (1) (2) (3) ())

- *What goes in the <??> ?!*

Subsets (2)

- Need to understand what this code is doing:

```
(define (subsets s)
  (if (null? s)
      (list nil)
      (let ((rest (subsets (cdr s))))
        (append rest (map <??> rest)))))
```

- Standard recursive design approach

- Solve the base case(s)
- Solve the recursive case in terms of **subsets**

- Base case: `()`

- List of all subsets of `()`: `(())` (a list containing the empty list)

Subsets (3)

- The code again:

```
(define (subsets s)
  (if (null? s)
      (list nil)
      (let ((rest (subsets (cdr s))))
        (append rest (map <??> rest)))))
```

- Recursive case:

- Generate all subsets of the rest of the list, then store that in the `rest` variable
- For example input `(1 2 3)`, `rest` contains subsets of `(2 3)`
 - `((2 3) (2) (3) ())`
- What must we do to take that result and generate final result?

Subsets (4)

- The code again:

```
(define (subsets s)
  (if (null? s)
      (list nil)
      (let ((rest (subsets (cdr s))))
        (append rest (map <??> rest))))))
```

- Recursive case, continued:

- The answer will contain all subsets of (2 3)...

 - e.g. ((2 3) (2) (3) ())

- Also need to incorporate (car s) into the results somehow

 - e.g. ((1 2 3) (1 2) (1 3) (1))
 - The purpose of the (map <??> rest) statement: add (car s) to front of each sublist in rest