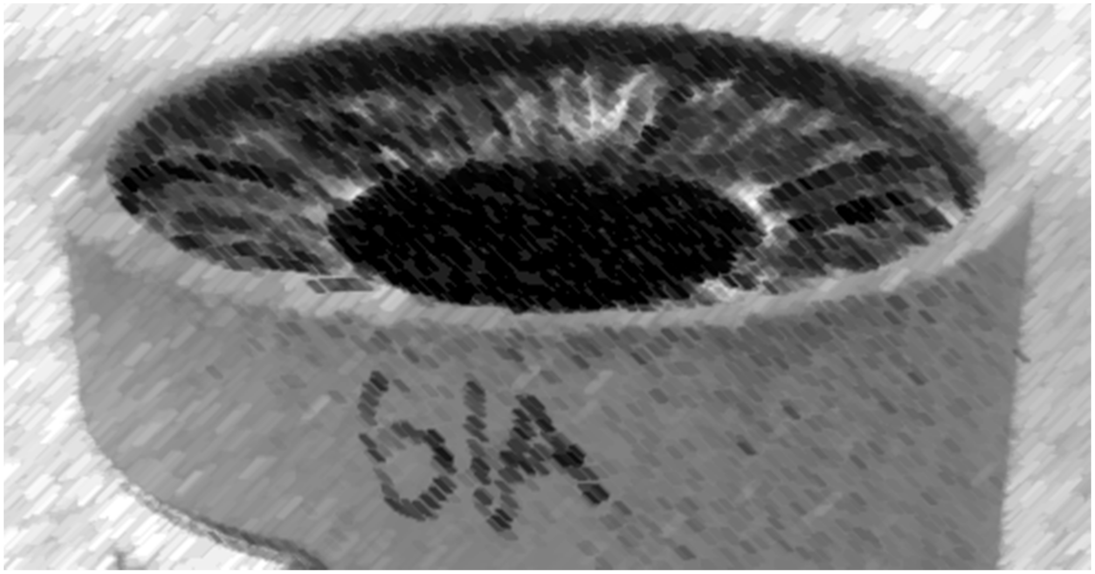


CS61A Lecture 31

Amir Kamil

UC Berkeley

April 3, 2013

Announcements

- HW9 due tonight
- Ants extra credit due tonight
    - See Piazza for submission instructions
- Hog revisions out, due Monday
- HW10 out tonight

Pairs

Scheme has built-in pairs that use weird names:

- **`cons`**: Two-argument procedure that **creates a pair**
- **`car`**: Procedure that returns the **first element** of a pair
- **`cdr`**: Procedure that returns the **second element** of a pair

A pair is represented by a dot between the elements, enclosed in parentheses

```
> (cons 1 2)
(1 . 2)
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
```

Recursive Lists

A recursive list can be represented as a pair in which the second element is a recursive list or the empty list

Scheme lists are recursive lists:

- **`nil`** is the empty list
- A non-empty Scheme list is a pair in which the second element is **`nil`** or a Scheme list

Scheme lists are written as space-separated combinations

```
> (define x (cons 1 (cons 2 (cons 3 (cons 4 nil)))))
> x
(1 2 3 4)
> (cdr x)
(2 3 4)
> (cons 1 (cons 2 (cons 3 4)))
(1 2 3 . 4)
```

Not a well-formed list!

Symbolic Programming

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
```

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
```

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
```

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

```
> (list 'a 'b)
```

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

```
> (list 'a 'b)
(a b)
```

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

```
> (list 'a 'b)
(a b)
> (list 'a b)
```

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Symbols are now values

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Symbols are now values

Quotation can also be applied to combinations to form lists

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Symbols are now values

Quotation can also be applied to combinations to form lists

```
> (car '(a b c))
```

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Symbols are now values

Quotation can also be applied to combinations to form lists

```
> (car '(a b c))
a
```

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Symbols are now values

Quotation can also be applied to combinations to form lists

```
> (car '(a b c))
a
> (cdr '(a b c))
```

Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
```

No sign of “a” and “b” in the resulting value

Quotation prevents something from being evaluated by Lisp

```
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

Symbols are now values

Quotation can also be applied to combinations to form lists

```
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

Scheme Lists and Quotation

Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
```

Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
```

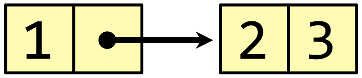
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
```



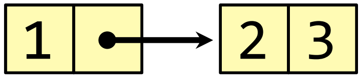
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
(1 2 . 3)
```



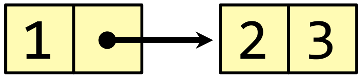
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
```



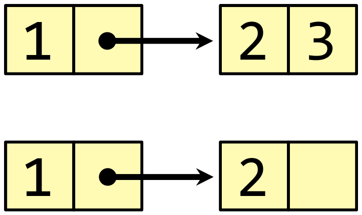
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
```



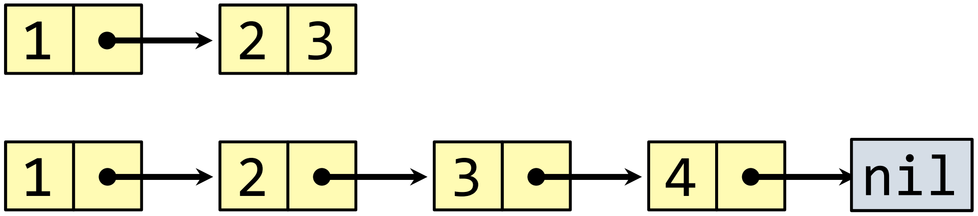
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
```



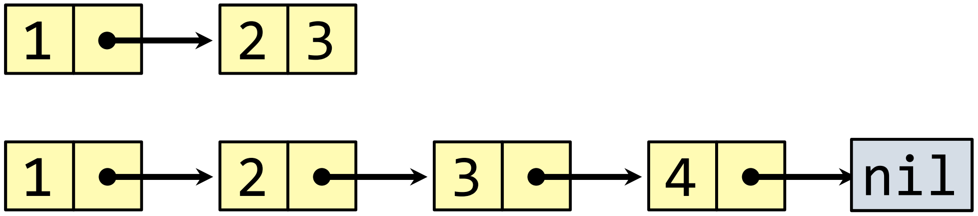
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
```



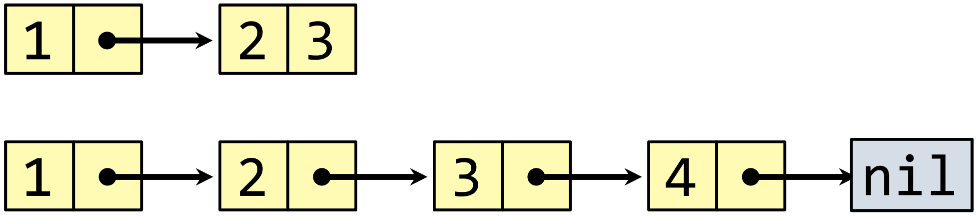
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
```



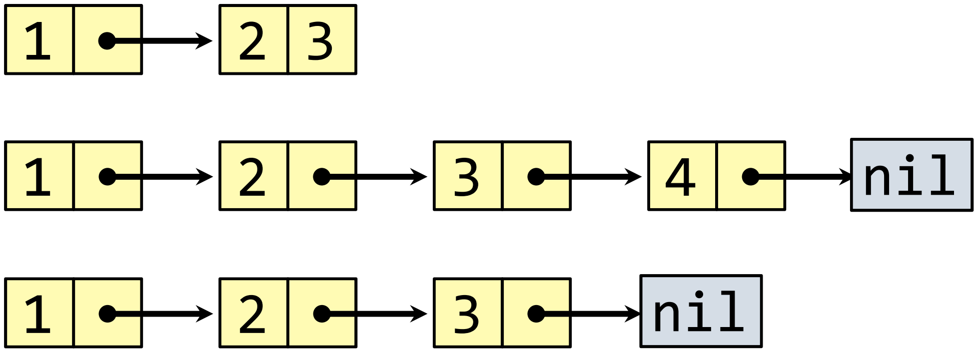
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
```



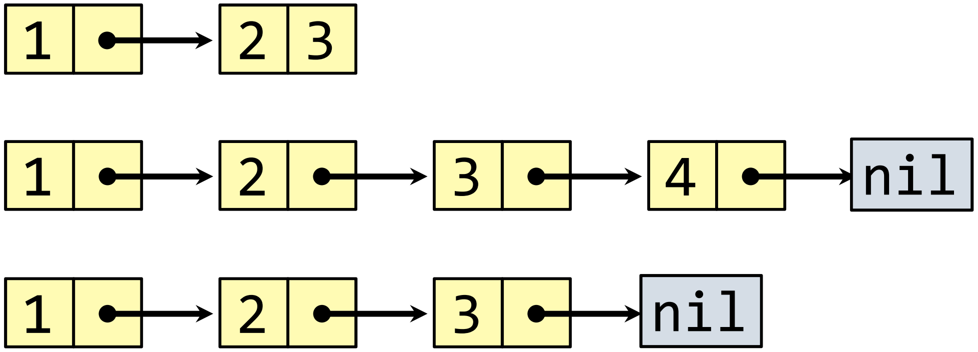
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)
```



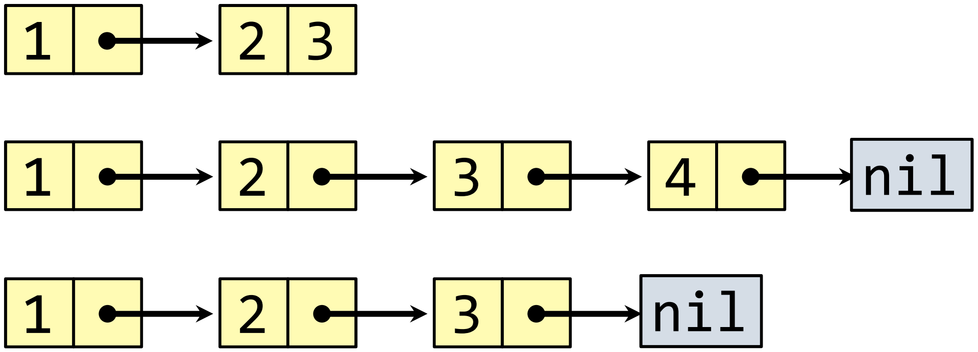
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)
```



What is the printed result of evaluating this expression?

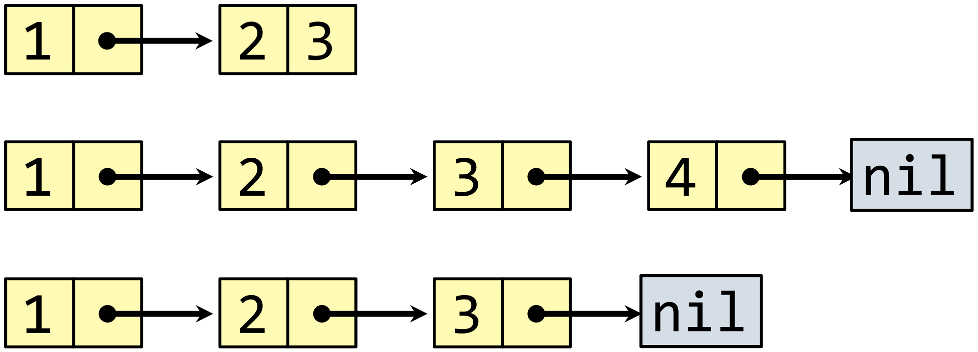
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)
```



What is the printed result of evaluating this expression?

```
> (cdr '((1 2) . (3 4 . (5))))
```

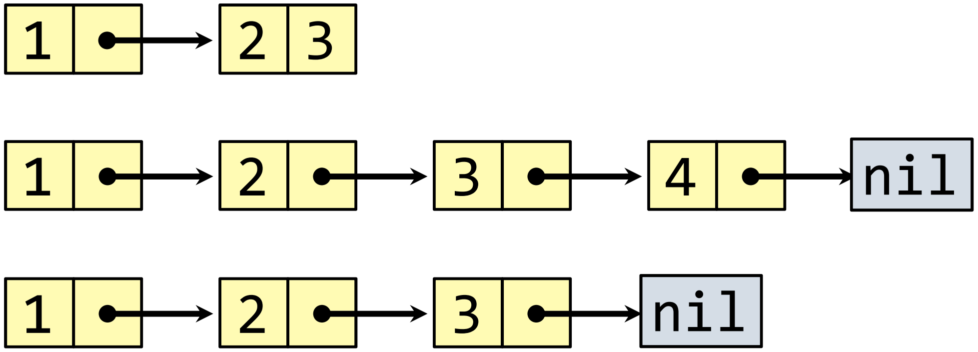
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

```
> (cdr (cdr '(1 2 . 3)))
3
```

However, dots appear in the output only of ill-formed lists

```
> '(1 2 . 3)
(1 2 . 3)
> '(1 2 . (3 4))
(1 2 3 4)
> '(1 2 3 . nil)
(1 2 3)
```



What is the printed result of evaluating this expression?

```
> (cdr '((1 2) . (3 4 . (5))))
(3 4 5)
```

The Let Special Form

Let expressions introduce a new frame, with the given bindings

The Let Special Form

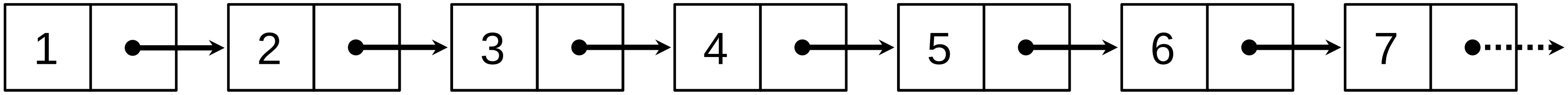
Let expressions introduce a new frame, with the given bindings

```
(let ((<name> <exp>) ...) <body>)
```

The Let Special Form

Let expressions introduce a new frame, with the given bindings

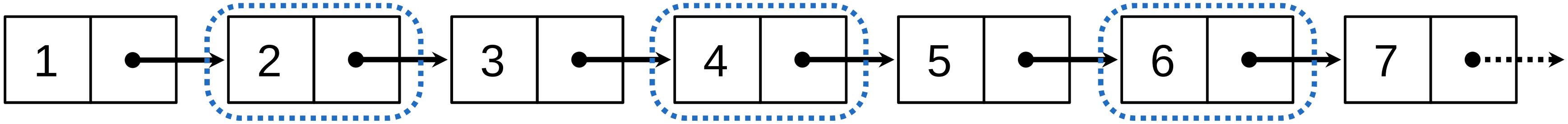
`(let ((<name> <exp>) ...) <body>)`



The Let Special Form

Let expressions introduce a new frame, with the given bindings

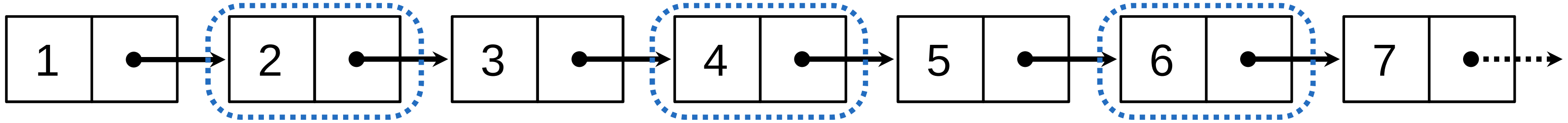
`(let ((<name> <exp>) ...) <body>)`



The Let Special Form

Let expressions introduce a new frame, with the given bindings

```
(let ((<name> <exp>) ...) <body>)
```

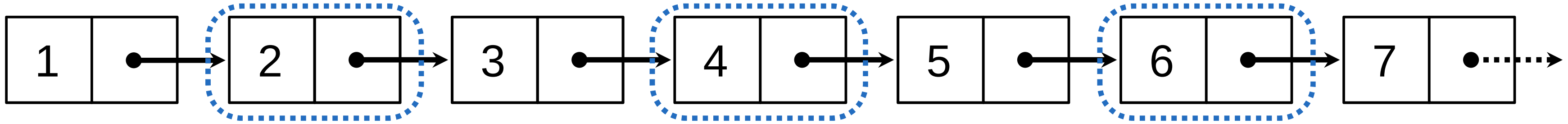


```
(define (filter fn s)
```

The Let Special Form

Let expressions introduce a new frame, with the given bindings

`(let ((<name> <exp>) ...) <body>)`

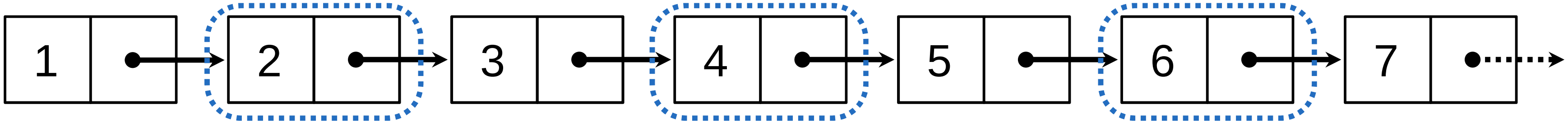


```
(define (filter fn s)
  (if (null? s)
```

The Let Special Form

Let expressions introduce a new frame, with the given bindings

```
(let ((<name> <exp>) ...) <body>)
```

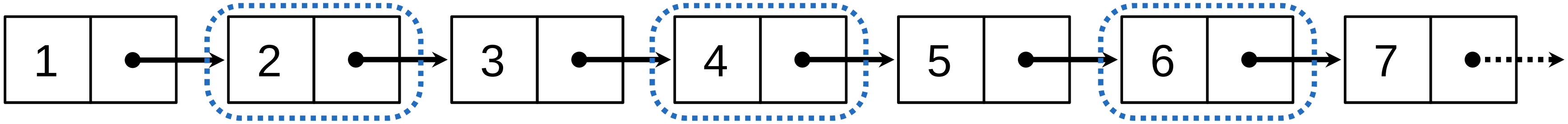


```
(define (filter fn s)
  (if (null? s)
      s
```

The Let Special Form

Let expressions introduce a new frame, with the given bindings

```
(let ((<name> <exp>) ...) <body>)
```

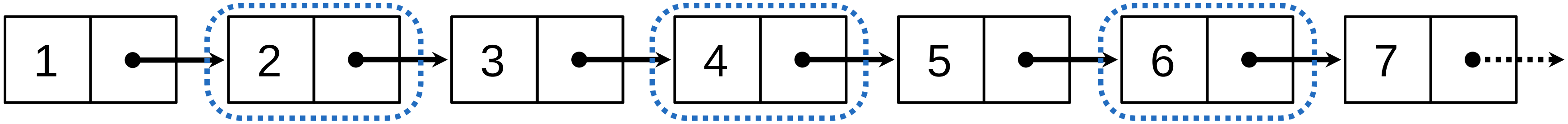


```
(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s))
```

The Let Special Form

Let expressions introduce a new frame, with the given bindings

```
(let ((<name> <exp>) ...) <body>)
```

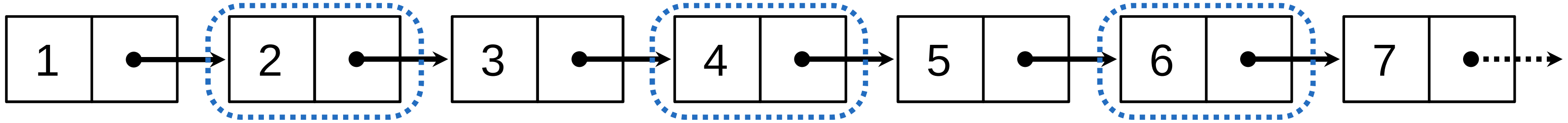


```
(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s))
            (rest (filter fn (cdr s))))
```

The Let Special Form

Let expressions introduce a new frame, with the given bindings

```
(let ((<name> <exp>) ...) <body>)
```

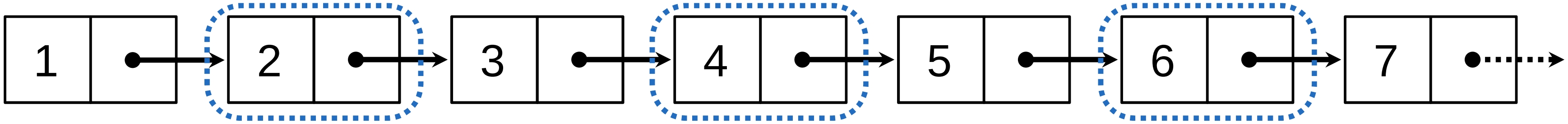


```
(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s))
            (rest (filter fn (cdr s))))
        (if (fn first)
```

The Let Special Form

Let expressions introduce a new frame, with the given bindings

```
(let ((<name> <exp>) ...) <body>)
```

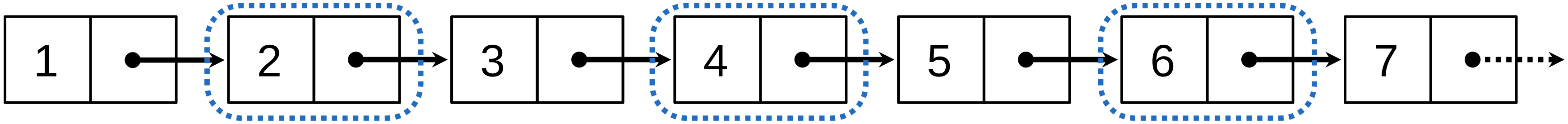


```
(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s))
            (rest (filter fn (cdr s))))
        (if (fn first)
            (cons first rest)
```

The Let Special Form

Let expressions introduce a new frame, with the given bindings

```
(let ((<name> <exp>) ...) <body>)
```

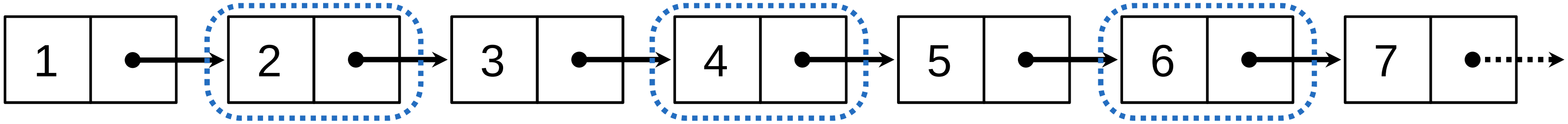


```
(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s))
            (rest (filter fn (cdr s))))
        (if (fn first)
            (cons first rest)
            rest))))
```

The Let Special Form

Let expressions introduce a new frame, with the given bindings

```
(let ((<name> <exp>) ...) <body>)
```



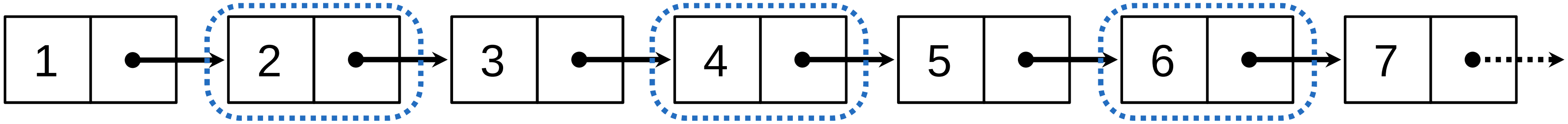
```
(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s))
            (rest (filter fn (cdr s))))
        (if (fn first)
            (cons first rest)
            rest))))
```

```
> (filter even? '(1 2 3 4 5 6 7))
```

The Let Special Form

Let expressions introduce a new frame, with the given bindings

```
(let ((<name> <exp>) ...) <body>)
```



```
(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s))
            (rest (filter fn (cdr s))))
        (if (fn first)
            (cons first rest)
            rest))))
```

```
> (filter even? '(1 2 3 4 5 6 7))
(2 4 6)
```

Quick Sort

Quick Sort

Quick sort algorithm:

Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)

Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot

Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece

Quick Sort

Quick sort algorithm:

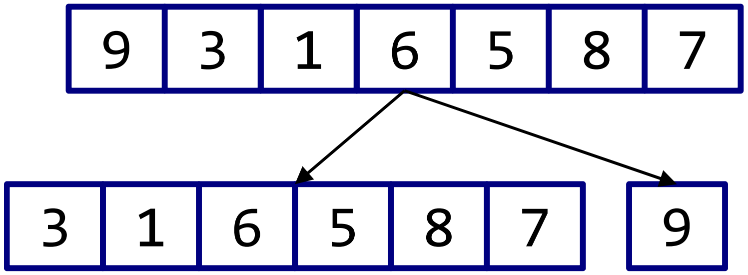
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece

9	3	1	6	5	8	7

Quick Sort

Quick sort algorithm:

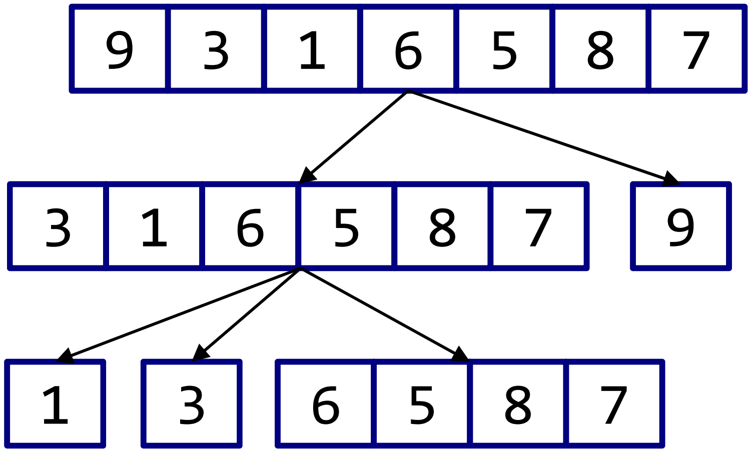
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece



Quick Sort

Quick sort algorithm:

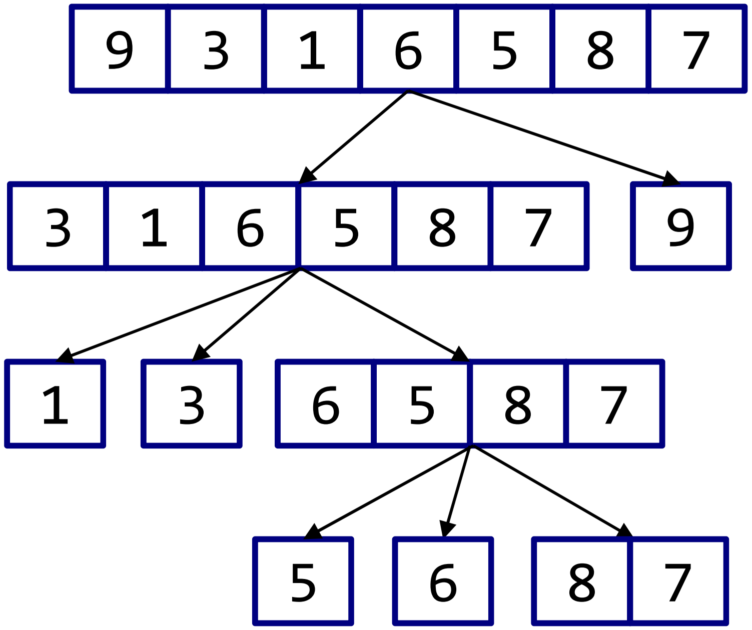
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece



Quick Sort

Quick sort algorithm:

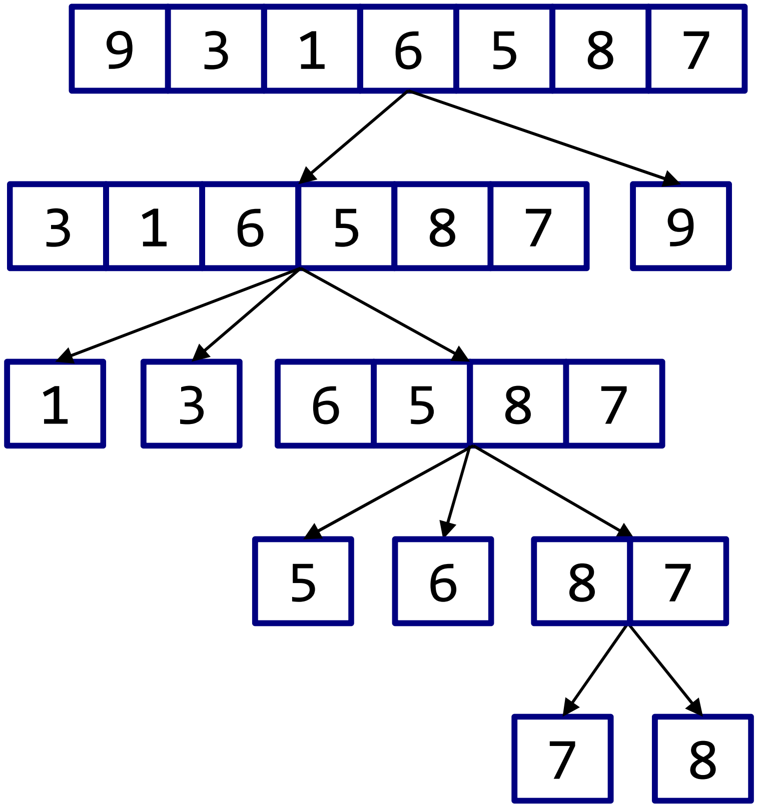
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece



Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece

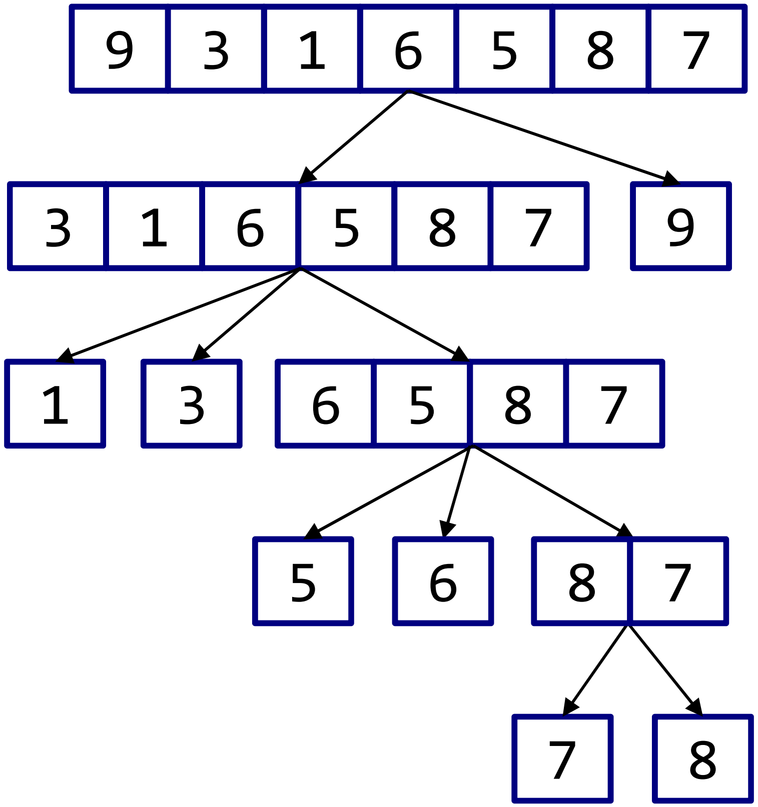


Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece

```
(define (filter-comp comp pivot s)
```

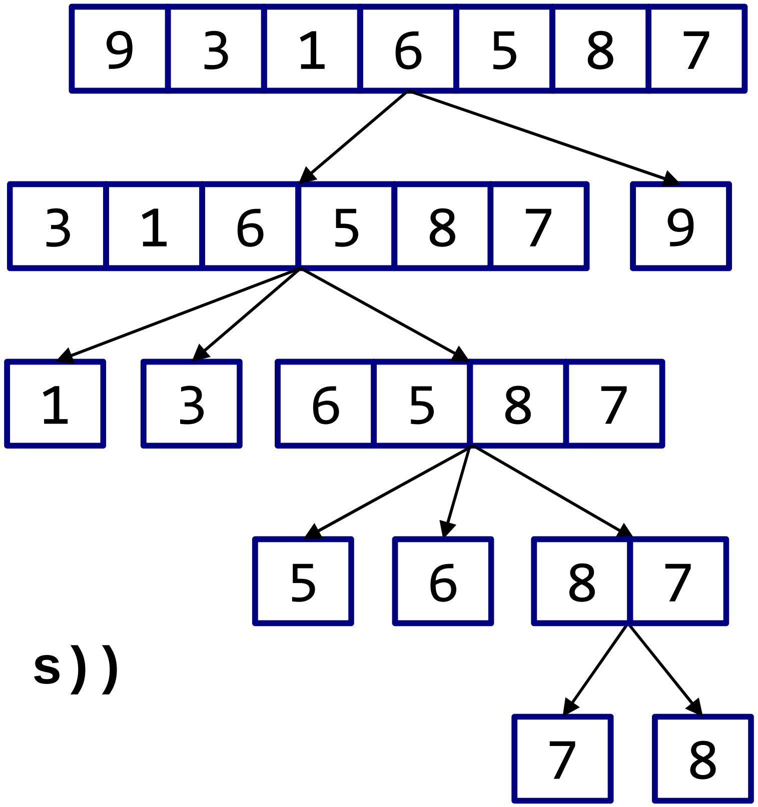


Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece

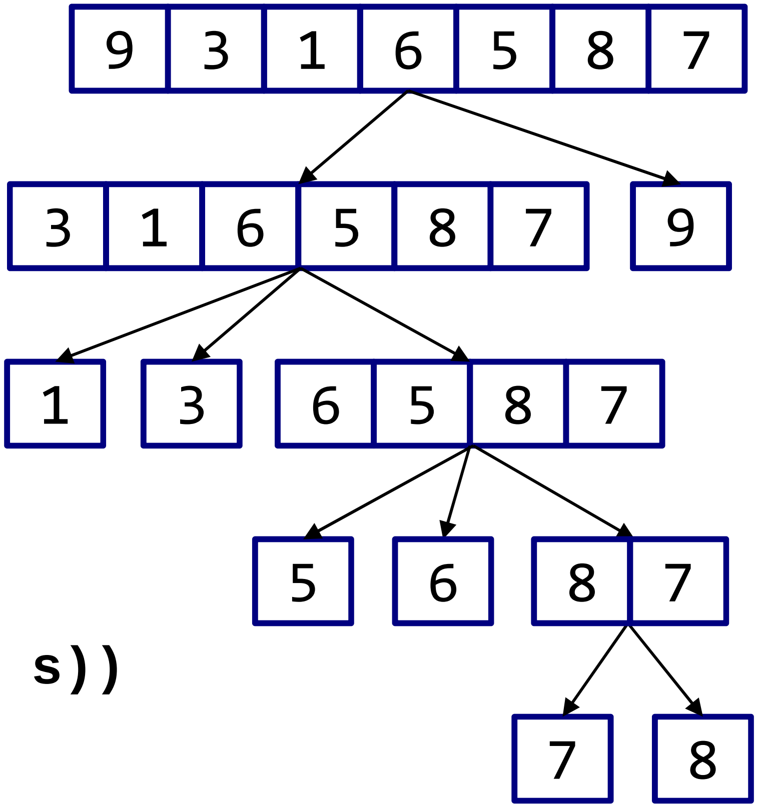
```
(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))
```



Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece



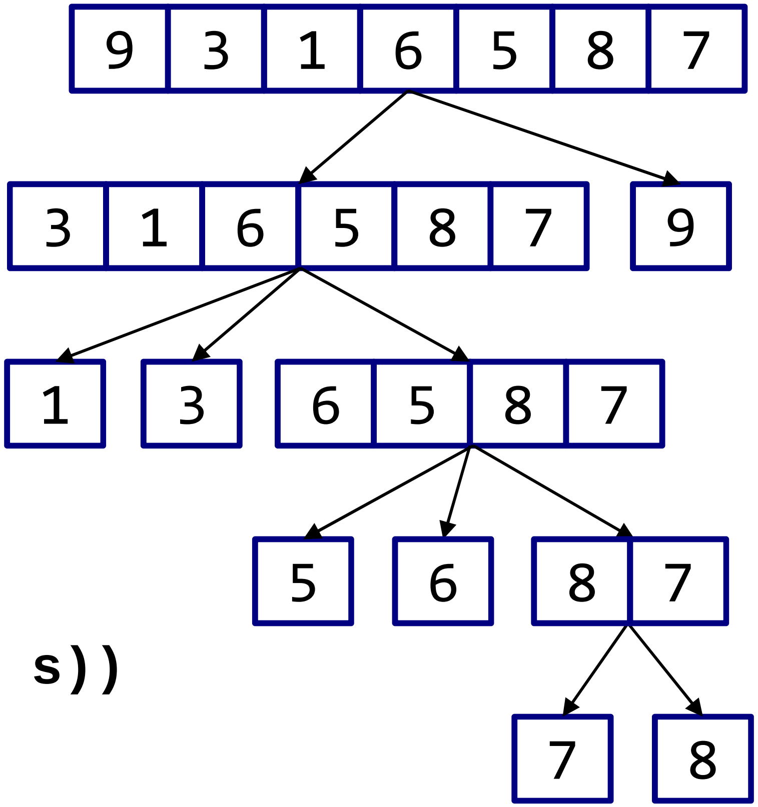
```
(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))

(define (quick-sort s)
```

Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece



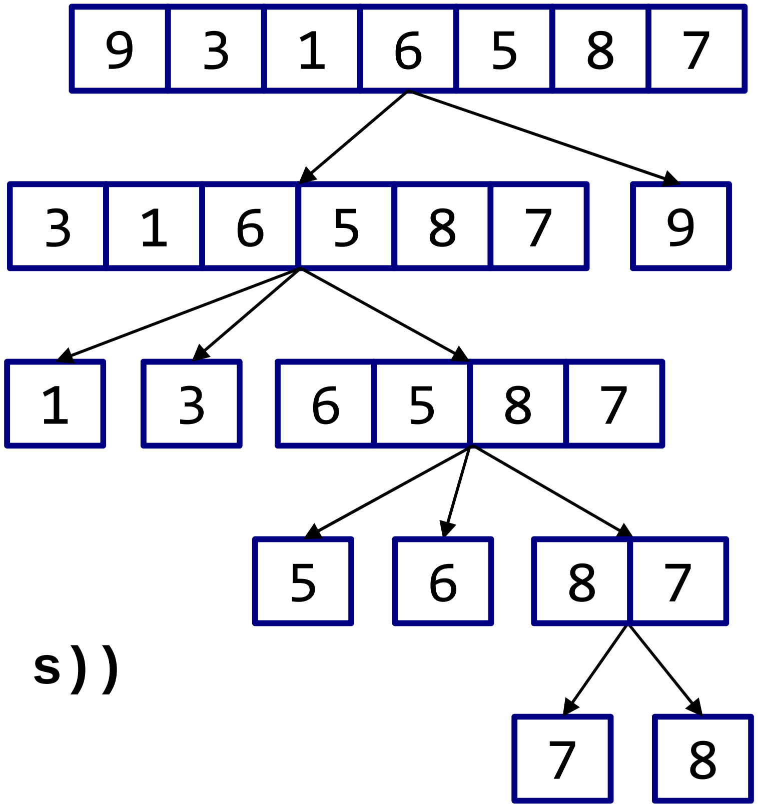
```
(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))

(define (quick-sort s)
  (if (<= (length s) 1)
```

Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece



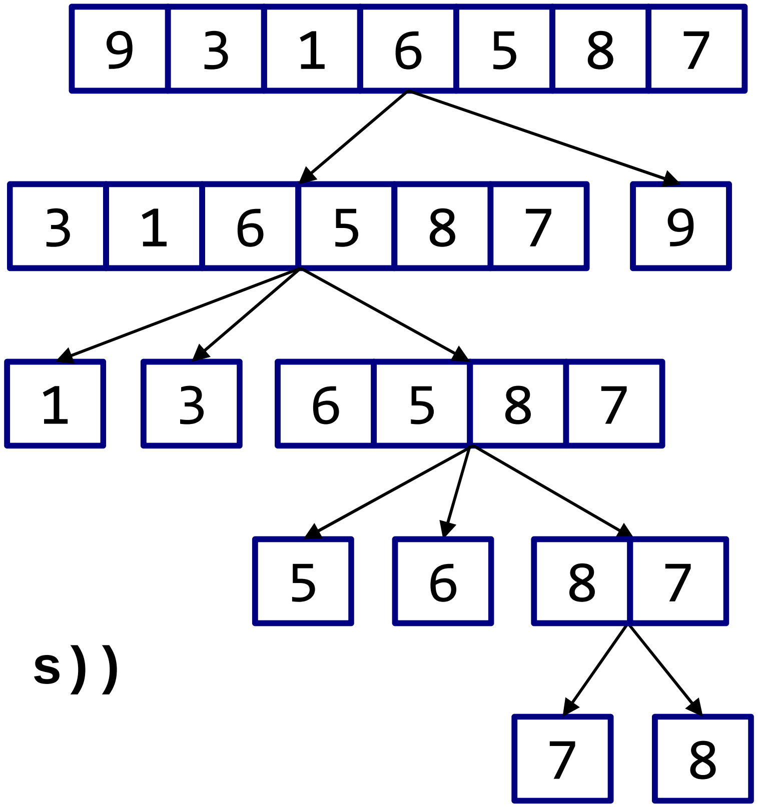
```
(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))

(define (quick-sort s)
  (if (<= (length s) 1)
      s
```

Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece



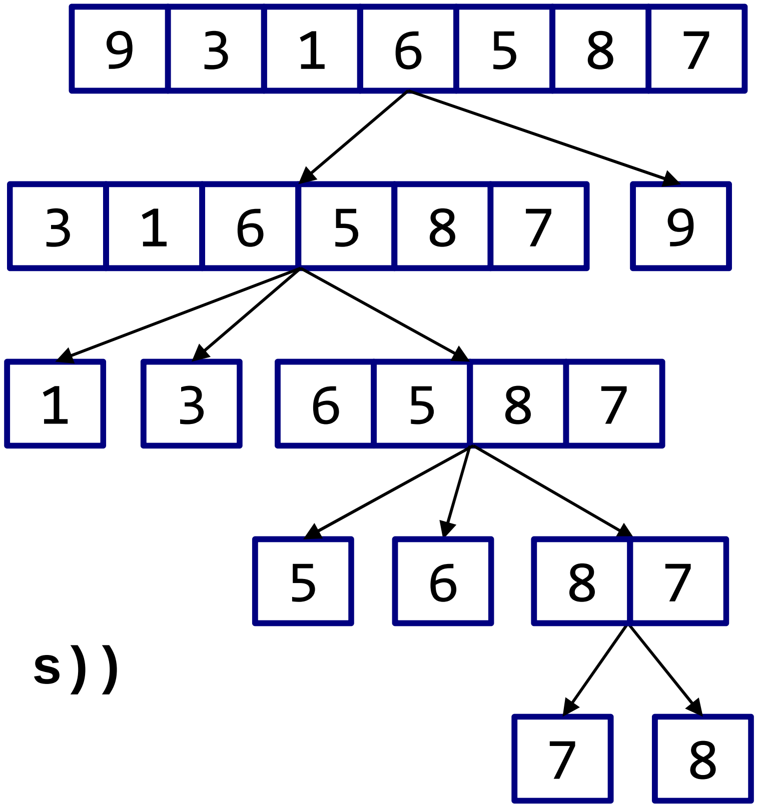
```
(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))

(define (quick-sort s)
  (if (<= (length s) 1)
      s
      (let ((pivot (car s)))
```

Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece



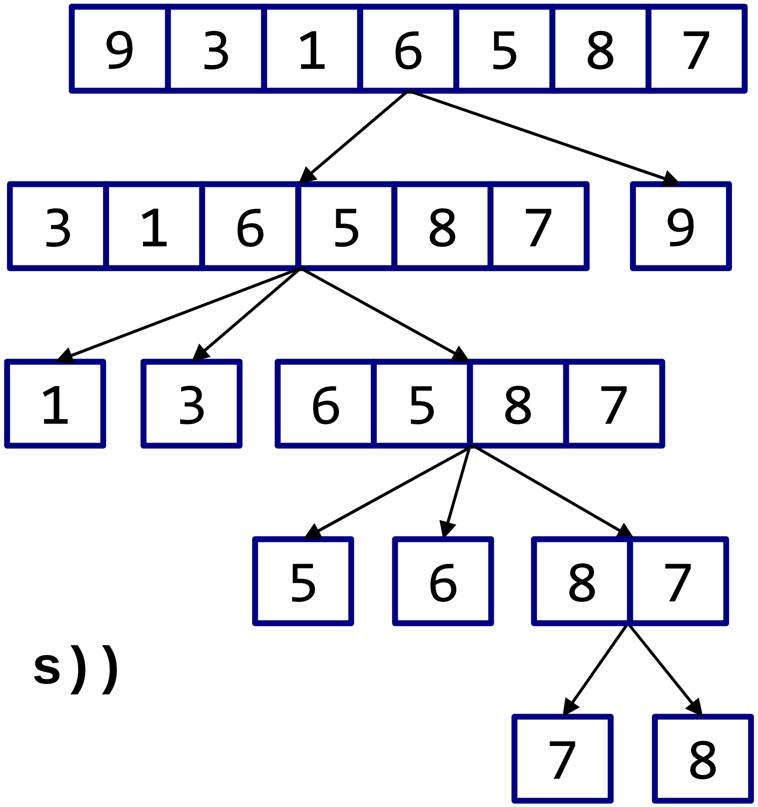
```
(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))

(define (quick-sort s)
  (if (<= (length s) 1)
      s
      (let ((pivot (car s)))
        (append (quick-sort (filter-comp < pivot s))
```

Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece



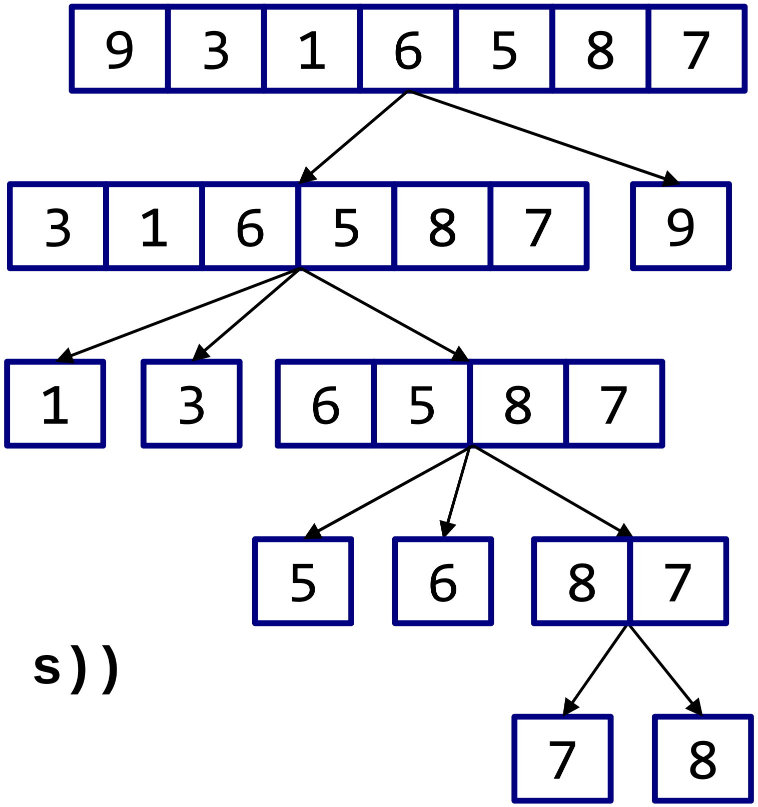
```
(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))

(define (quick-sort s)
  (if (<= (length s) 1)
      s
      (let ((pivot (car s)))
        (append (quick-sort (filter-comp < pivot s))
                (filter-comp = pivot s)
```

Quick Sort

Quick sort algorithm:

1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece



```
(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))

(define (quick-sort s)
  (if (<= (length s) 1)
      s
      (let ((pivot (car s)))
        (append (quick-sort (filter-comp < pivot s))
                (filter-comp = pivot s)
                (quick-sort (filter-comp > pivot s))))))
```

The Begin Special Form

Begin expressions allow sequencing

The Begin Special Form

Begin expressions allow sequencing

$$(\texttt{begin <exp}_1\texttt{> <exp}_2\texttt{> ... <exp}_n\texttt{>})$$

The Begin Special Form

Begin expressions allow sequencing

(**begin** $<exp_1>$ $<exp_2>$ ... $<exp_n>$)

```
(define (repeat k fn)
```

The Begin Special Form

Begin expressions allow sequencing

$$\texttt{(begin <exp}_1\texttt{> <exp}_2\texttt{> ... <exp}_n\texttt{>)}$$

```
(define (repeat k fn)
  (if (> k 0)
```

The Begin Special Form

Begin expressions allow sequencing

```
(begin <exp₁> <exp₂> ... <expₙ>)
```

```
(define (repeat k fn)
  (if (> k 0)
      (begin (fn) (repeat (- k 1) fn))
```

The Begin Special Form

Begin expressions allow sequencing

$$\texttt{(begin <exp}_1\texttt{> <exp}_2\texttt{> ... <exp}_n\texttt{>)}$$

```
(define (repeat k fn)
  (if (> k 0)
      (begin (fn) (repeat (- k 1) fn))
      'done))
```

The Begin Special Form

Begin expressions allow sequencing

$$\texttt{(begin <exp}_1\texttt{> <exp}_2\texttt{> ... <exp}_n\texttt{>)}$$

```
(define (repeat k fn)
  (if (> k 0)
      (begin (fn) (repeat (- k 1) fn))
      'done))

(define (tri fn)
```

The Begin Special Form

Begin expressions allow sequencing

(begin <exp$_1$> <exp$_2$> ... <exp$_n$>)

```
(define (repeat k fn)
  (if (> k 0)
      (begin (fn) (repeat (- k 1) fn))
      'done))

(define (tri fn)
  (repeat 3 (lambda () (fn) (lt 120))))
```

The Begin Special Form

Begin expressions allow sequencing

$$\texttt{(begin <exp}_1\texttt{> <exp}_2\texttt{> ... <exp}_n\texttt{>)}$$

```
(define (repeat k fn)
  (if (> k 0)
      (begin (fn) (repeat (- k 1) fn))
      'done))

(define (tri fn)
  (repeat 3 (lambda () (fn) (lt 120))))

(define (sier d k)
```

The Begin Special Form

Begin expressions allow sequencing

```
(begin <exp₁> <exp₂> ... <expₙ>)
```

```
(define (repeat k fn)
  (if (> k 0)
      (begin (fn) (repeat (- k 1) fn))
      'done))

(define (tri fn)
  (repeat 3 (lambda () (fn) (lt 120))))

(define (sier d k)
  (tri (lambda () (if (= k 1) (fd d) (leg d k)))))
```

The Begin Special Form

Begin expressions allow sequencing

```
(begin <exp1> <exp2> . . . <expn>)
```

```
(define (repeat k fn)
  (if (> k 0)
      (begin (fn) (repeat (- k 1) fn))
      'done))

(define (tri fn)
  (repeat 3 (lambda () (fn) (lt 120))))

(define (sier d k)
  (tri (lambda () (if (= k 1) (fd d) (leg d k)))))

(define (leg d k)
```

The Begin Special Form

Begin expressions allow sequencing

$$\texttt{(begin <exp}_1\texttt{> <exp}_2\texttt{> ... <exp}_n\texttt{>)}$$

```
(define (repeat k fn)
  (if (> k 0)
      (begin (fn) (repeat (- k 1) fn))
      'done))

(define (tri fn)
  (repeat 3 (lambda () (fn) (lt 120))))

(define (sier d k)
  (tri (lambda () (if (= k 1) (fd d) (leg d k)))))

(define (leg d k)
  (sier (/ d 2) (- k 1)) (penup) (fd d) (pendown))
```