Functional Programming II

Spring 2014

Carola Wenk

List Manipulation

- Of course, more sophisticated algorithms will require us to access parts of a list.

- The `cons` function prepends an element to a list.

- The `car` function returns the first element of a list.

- The `cdr` function removes the first element of a list, and returns the remaining list.

- These basic functions are used to implement all of the list operations we've seen (e.g. indexing and slicing), and many of these are implemented in the Scheme standard library.

List Manipulation

List functions in the Scheme standard library:

- The `list` function returns a list of objects.
- The `length` function returns the length of a list.
- The `append` function concatenates multiple lists.
- The `reverse` function reverses a list.
- One can combine up to four `car` and `cdr` in one function: The (`cadr L`) function is short for `(car (cdr L))`

Variable Binding

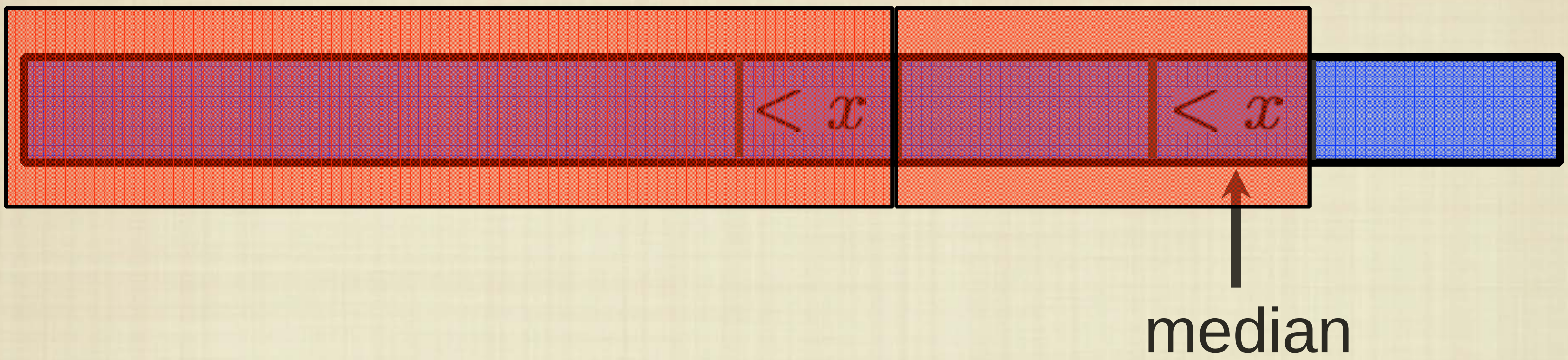
```
(let ([<var> <binding>] ... ) <body>)
```

- ```
  (let ([a 5])
    (+ a 7))
  ```
  evaluates to: 12

- ```
  (let ([a 5] [b 7])
    (+ a b))
  ```
  evaluates to: 12

- This works for functions as well:
  ```
  (let ([f +])
    (f 1 2))
  ```
  evaluates to: 3

Searching A Sorted List



<u>Binary Search:</u>

1. Test whether the x is less than the median (if it is equal, we are done).

2. Recursively search the half of the list that x is in.

3. We are done when the “correct” side of the list is empty.

Binary Search in Scheme

- The structure of the Scheme function clearly corresponds to our previous recursive implementations:

```
(define (bsearch L x)
  (let ([mid (quotient (length L) 2)])
  (if (equal? L '())
      #f
      (cond ((= x (list-ref L mid))
             #t)
            ((< x (list-ref L mid))
             (bsearch (take L mid) x))
            (else
             (bsearch (drop L (+ 1 mid)) x))))))
```

How long does this algorithm take to execute?

It actually depends - how long does `list-ref` take?

Binary Search in Scheme

- The structure of the Scheme function clearly corresponds to our previous recursive implementations:

```
(define (bsearch L x)
  (let ([mid (quotient (length L) 2)])
  (if (equal? L '())
      #f
      (cond ((= x (list-ref L mid))
             #t)
            ((< x (list-ref L mid))
             (bsearch (take L mid) x))
            (else
             (bsearch (drop L (+ 1 mid)) x))))))
```

Interestingly, we don't really know!

If the list operations take constant time, then this implementation is logarithmic in the list size.

Otherwise?

Correctness

```
(define (bsearch L x)
  (let ([mid (quotient (length L) 2)])
  (if (equal? L '())
       #f
       (cond ((= x (list-ref L mid))
               #t)
              ((< x (list-ref L mid))
               (bsearch (take L mid) x))
              (else
               (bsearch (drop L (+ 1 mid)) x))))))
```

Base Case: If the input list is empty, then clearly `x` is not in it.

Inductive Step: Let n be the length of the input list `L`, and suppose that bsearch works correctly for any list of length less than n. Now, consider two cases: either `x` is the median of `L`, or it isn't. If it is, our code correctly handles that case and we're done. If not, `bsearch` will be called on a list with fewer than n elements, and we're done.