CS3:
Introduction to Symbolic Programming

Lecture 5:
Recursion

Spring 2007

Nate Titterton
nate@berkeley.edu

Schedule

5	Feb 12-16	Lecture: Introduction to Recursion Reading: Simply Scheme, Chapter 11 DbD case study, recursive version Lab: Recursion
6	Feb 19-23	Lecture: <*holiday*> Reading: "Roman Numerals" case study Special Lecture: Recursion, Midterm (Tuesday, Feb 20, 9:30-10:30, 306 Soda) Lab: More recursion
7	Feb 26 - Mar 2	Lecture: *Midterm 1* Lab: Advanced recursion
8	Mar 5 – 9	Lecture: Recursion Lab: Miniproject #2

Announcements

- **Nate's office hours:**
  - **Wednesday, 2-4, in 329 Soda**

- **Reading for this week**
  - **Simply Scheme, chapter 11**
  - **Difference between Dates, Recursive version**
  - **(These *will* be on the midterm)**

- **More reading next week…**

- **The last day to drop is Feb 16th**

Midterm 1: Feb 26th (in two weeks)

- **Location: ???**
  - Will send out via email, ucwise announcments
- **Time: In the lecture slot, plus 20 minutes**
  - (5:10-6:30)
- **Open book, open notes.**
  - Nothing that can compute, though
- **Everything we've covered, including the coming TWO weeks on recursion.**
  - (But not the "roman numerals" case study)

Special midterm issues

- **Special Lecture**
  - Tuesday, Feb 20th, 9:30-10:30, 306 Soda
- **TA-led review session: Date and Loc TBD**
  - Probably Sat, Feb 24, 2-4pm, 430 Soda
- **There are two practice exams in your reader**
  - The first is shorter than yours will be, the second is the right length
  - Do these as if you were taking a Midterm: i.e., in one sitting, without STk, etc.
- **Check the announcements for solutions, and more practice exams.**

Any questions about the miniproject?

Recursion

An algorithmic technique where a function, in order to accomplish a task, calls itself with some part of the task.

Using recursive procedures

- **Everyone thinks it's hard!**
  - **(well, it is… aha!-hard, not complicated-hard)**
- **Using repetition and loops to find answers**
- **The first technique (in this class) to handle arbitrary length inputs.**
  - **There are other techniques, easier for some problems.**

All recursion procedures need…

1. **Base Case (s)**
   - Where the problem is simple enough to be solved directly
2. **Recursive Cases (s)**
   1. **Divide the Problem**
      - into one or more smaller problems
   2. **Invoke the function**
      - Have it call itself recursively on each smaller part
   3. **Combine the solutions**
      - Combine each subpart into a solution for the whole

Problem: *find the first even number in a sentence of numbers*

```
(define (find-first-even sent)
  (if <test>
      (first sent)

      (find-first-even (bf sent))
      <do the base case>
      <do the recursive case>
      ))
```

```
(define (find-first-even sent)
  (if (even? (first sent))
      (first sent)          ;base case: return
                            ; that even number
      (find-first-even (bf sent))
                            ;recurse on the
                            ; rest of sent
      ))
```

Count the number of words in a sentence

```
(define (count sent)

  (if (empty? (bf sent))

      1                          ;base case: return 1

      (+ 1
        (count (bf sent)))  ;recurse on the
                                 ; rest of sent

))
```

Base cases can be tricky

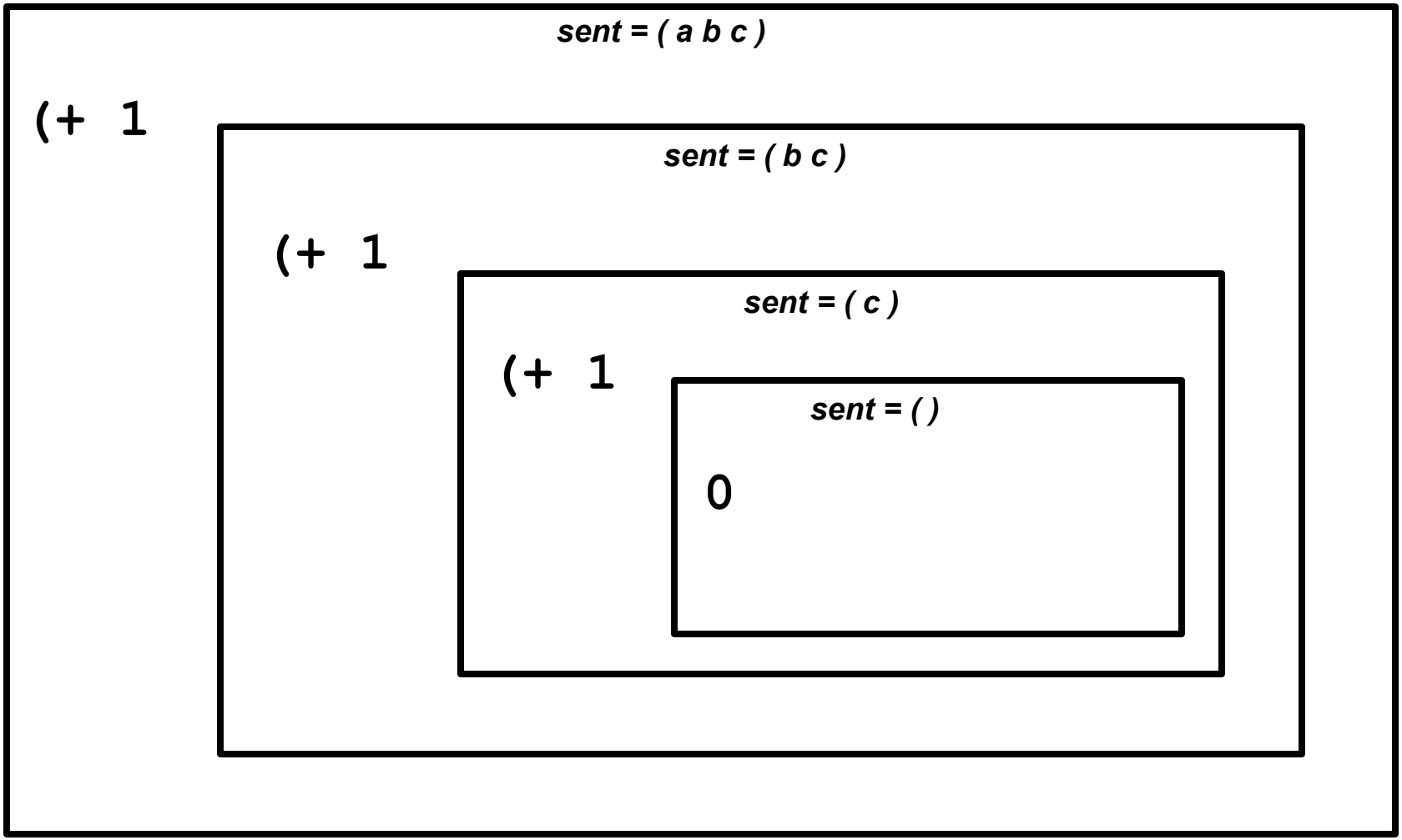
- **By checking whether the `(bf sent)` is empty, rather than `sent`, we won't choose the recursive case correctly on that last element!**
  - **Or, we need two base cases, one each for the last element being odd or even.**
- **Better: let the recursive cases handle *all* the elements**

Your book describes this well

Count the number of words in a sentence

```
(define (count sent)

  (if (empty? (bf sent)) )

      0                          ;base case: return 0

      (+ 1
        (count (bf sent))  ;recurse on the
                                 ; rest of sent

))
```

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



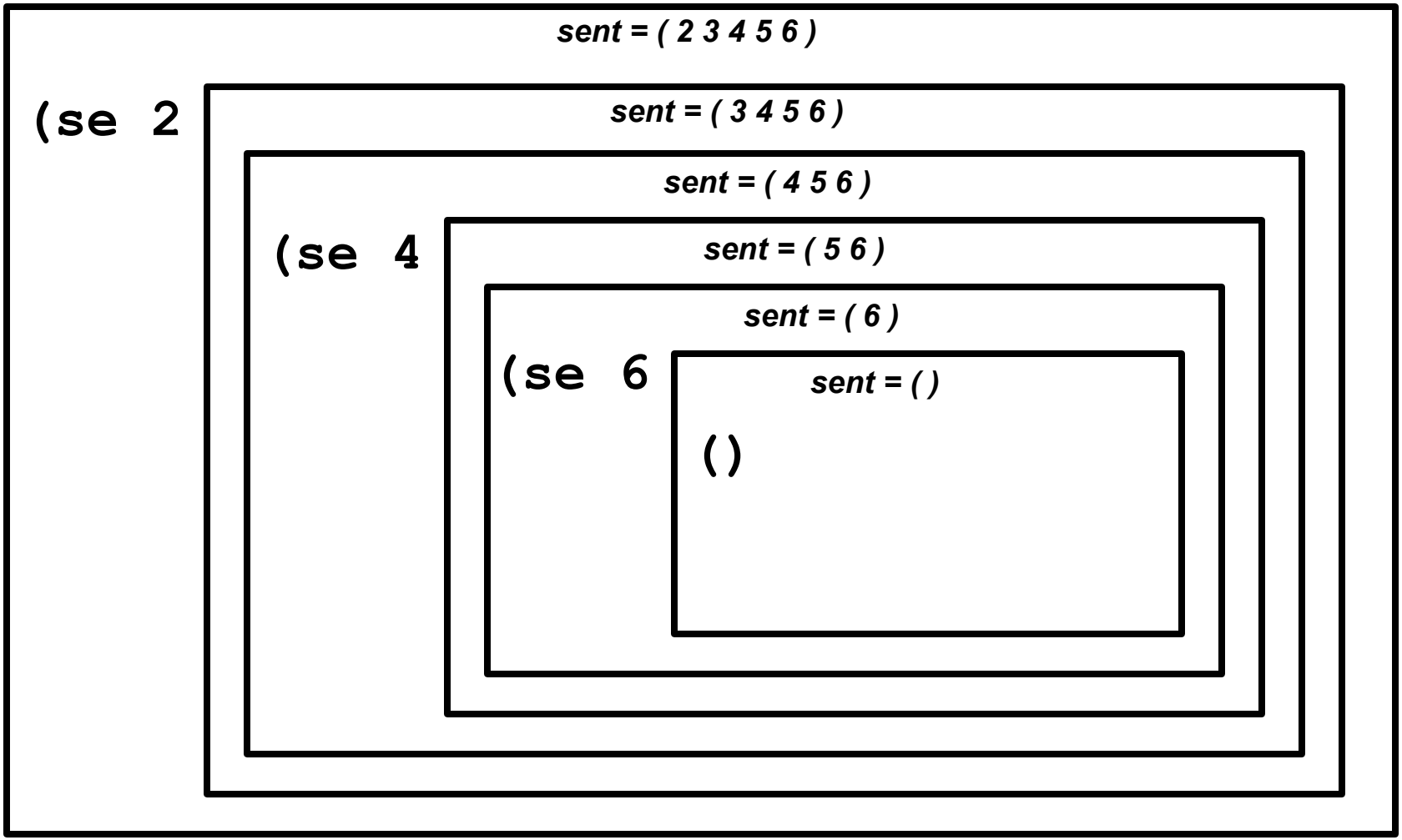
➔ (+ 1 (+ 1 (+ 1 0)))

➔ 3

Problem: *find all the even numbers in a sentence of numbers*

```
(define (find-evens sent)
  (cond ((empty? sent)          ;base case
         '()          )
        ((odd? (first sent)) ;rec case 1: odd
         (find-evens (bf sent)) )
        (else                   ;rec case 2: even
         (se (first sent)
             (find-evens (bf sent))) )
        ))
```

```
> (find-evens '(2 3 4 5 6))
```



➔ (se 2 (se 4 (se 6 ())))

➔ (2 4 6)