CS3:

Introduction to Symbolic Programming

Lecture 4:
“DbD” and data abstraction;
Introduction to Recursion

Spring 2008

Nate Titterton
nate@berkeley.edu

Schedule

3	Feb 4-Feb 8	Lecture: Conditionals, Case Studies Lab: “Difference between Dates”, MP#1
4	Feb 11-15	Lecture: DbD, recursion Reading (Thur/Fri): Simply Scheme chap. 11 Lab: (1) Miniproject 1 (2) Introduction to recursion
5	Feb 18-22	Lecture: *Holiday, no lecture* Reading: “DbD” case study, recursive version Simply Scheme, Chapter 12 (for Tue/Wed) “Roman Numerals” case study (for Thur/Fri) Lab: More complex recursion
6	Feb 25-29	Lecture: *Midterm #1* Lab: Recursion with multiple arguments

Announcements

- Nate's office hours (this week):
  - Wed, 2-4, 329 Soda
- Reading for this week
  - Simply Scheme, chapter 11
  - You need to do this before Lab on Thur/Fri
- Note: you need to take quizzes in the lab room
  - You are allowed 4 quizzes taken while not in attendance
- The last day to drop is approaching…
- If you have a MS Vista computer
  - There are new instructions up on getting Emacs and STk working on your local computer
  - Go to 333 Soda if you have trouble. Go anyway.

Midterm #1 is coming

- Midterm 1 is in 2 weeks (Feb 25th)
  - 90 minutes long (4:10-5:40)
  - It will not be in this room! Rather, 2050 Valley life sciences
  - Open book, open notes, no computers…
  - There is no lecture next Monday.
    - I plan on holding a make-up lecture on Wednesday afternoon, in lieu of office hours. I hope.
    - I'll post information to the course portal.
  - There will be a TA-led review session the weekend before.

Any questions about the miniproject?

Abstraction

“the process of leaving out consideration of one or more properties of a complex object or process so as to attend to others”

- **Abstracting with a new function**

  Using helper functions, basically…

  `(square x)` instead of `(* x x)`

  `(third sent)` instead of `(first (bf (bf sent)))`

- **Abstracting a new datatype**

  A datatype provides functionality necessary to store* "something" *important to the program

  - *Selectors*: **to look at parts of the "something".**
  - *Constructors:* **to create a new "something".**
  - *Tests* **(sometimes): to see whether you have a "something", or a "something else"**

Data abstration: words and sentences

Constructors: procedures to make a piece of data

- `word`, `sentence`

Selectors: procedures to return parts of that data piece

- `first`, `butfirst`, etc.

Tests: predicates that tell you which type of data you have

- `word?`, `sentence?`

Benefits

- Why is "leaving out consideration of", or "not knowing about", a portion of the program a good thing?

- Consider two ways one can "understand a program":
  - Knowing what each function does
  - Knowing what the inputs are (can be), and what the outputs are (will be).

Data abstraction in the DbD code

- How does the code separate out processing of the date-format from the logic that does the "real" work?

  - Selectors
    - month-name   (takes a date)
    - date-in-month (takes a date)
    - ? month-number (takes a month name)
  - Constructors? Tests?

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 (even? (first sent))

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

```
  (if <test>

     (<do the base case>)
     (<do the recursive case>)
```

Count the number of words in a sentence

```
(define (count sent)

  (if (empty? (bf sent))  ;last one?

     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

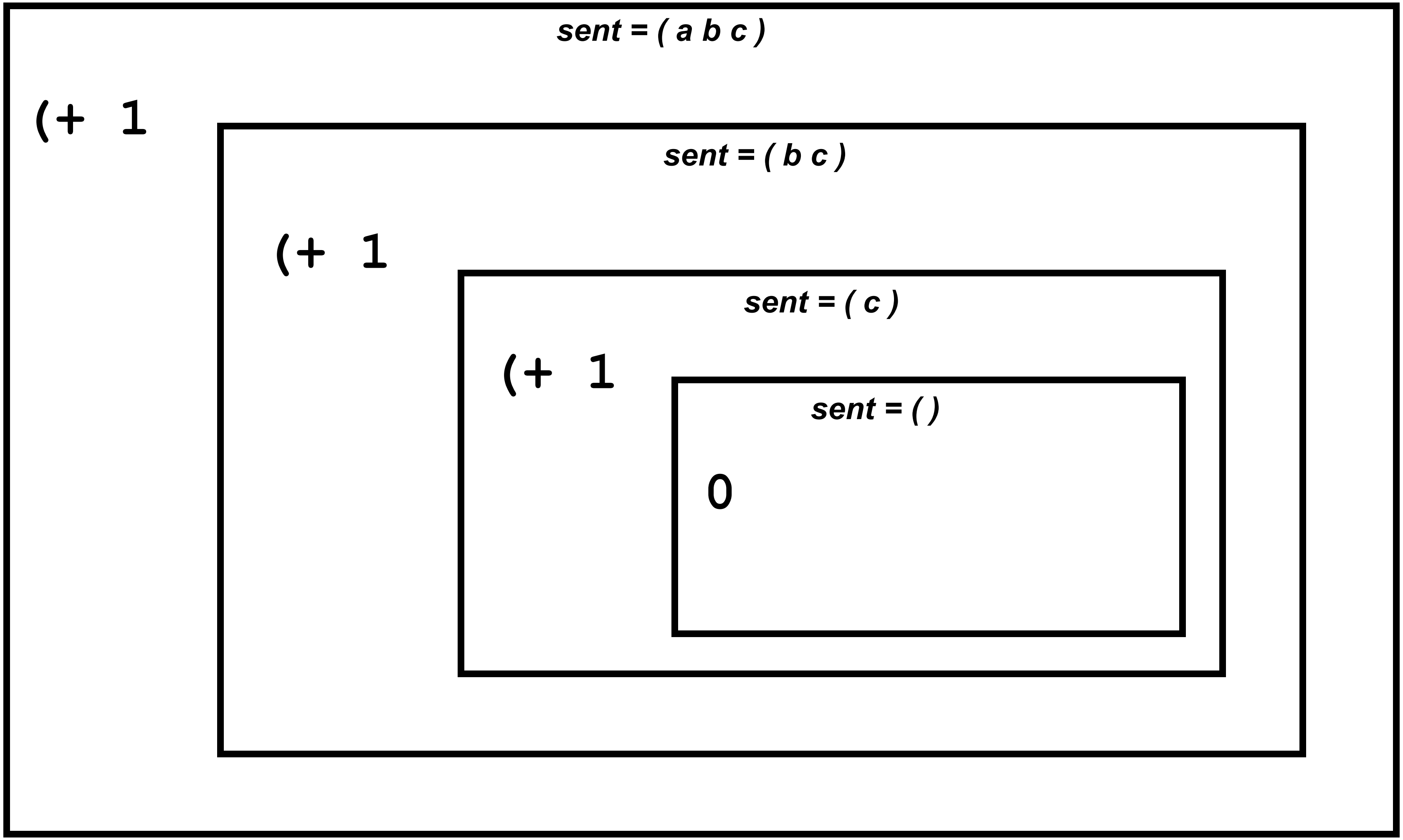
Count the number of words in a sentence

```
(define (count sent)

  (if (empty? (bf sent))  ;last one?

     1                    ;base case: return 1

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

))
```

Count the number of even-numbers

```
(define (count-evens sent)

  (cond ((empty? sent)  ;empty?
         0 )            ;base case: return 0

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

        ((odd? (first sent)
         (+ 0
           (count (bf sent))) ;recurse on the
                              ; rest of sent

   ))
```

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



sent = (a b c)

```
(+ 1
```

sent = (b c)

```
(+ 1
```

sent = (c)

```
(+ 1
```

sent = ( )

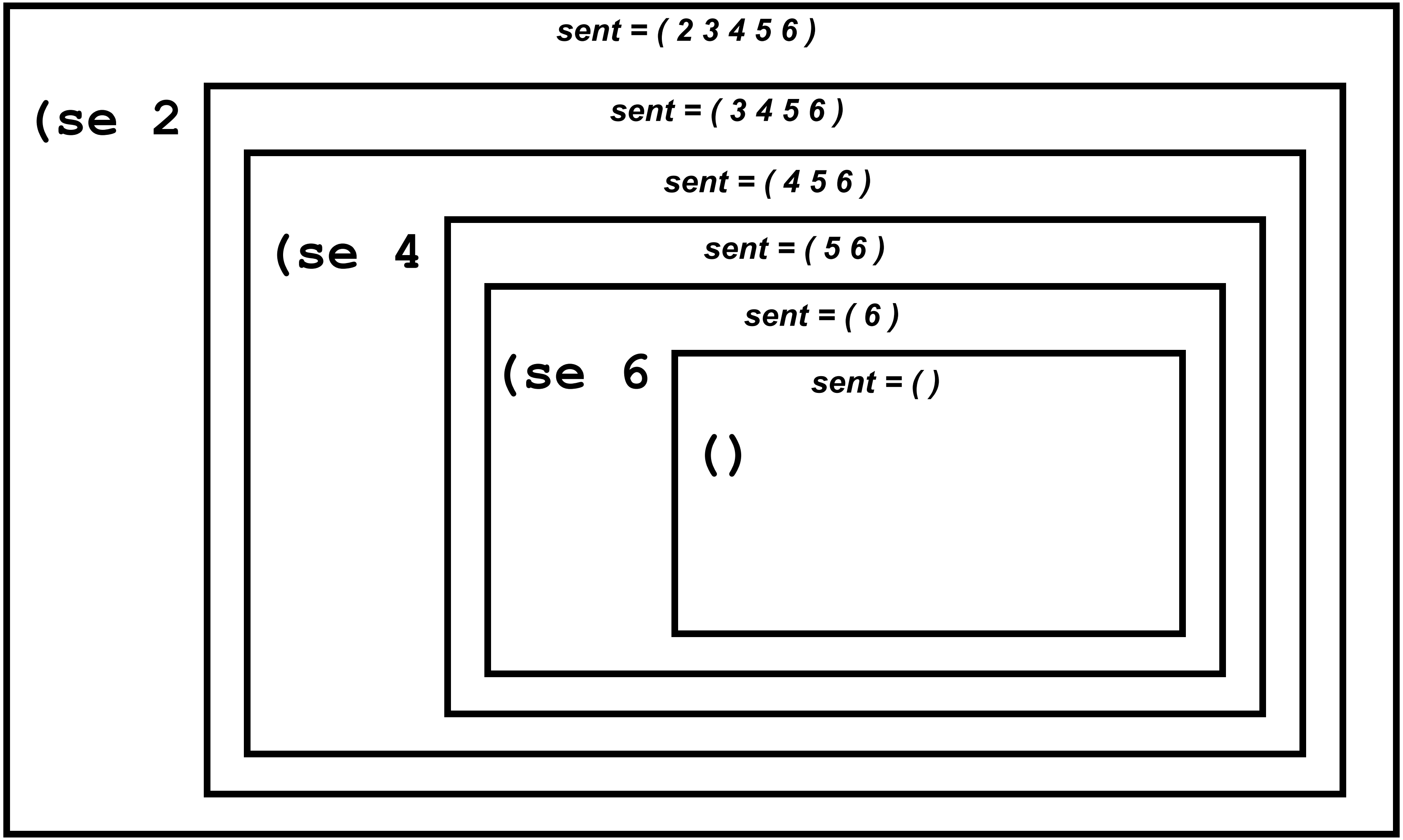
```
0
```

```
➔ (+ 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))
```



sent = (2 3 4 5 6)

```
(se 2
```

sent = (3 4 5 6)

sent = (4 5 6)

```
(se 4
```

sent = (5 6)

sent = (6)

```
(se 6
```

sent = ( )

```
()
```

```
➔ (se 2 (se 4 (se 6 ())))
➔ (2 4 6)
```