CS3:
Introduction to Symbolic Programming

Lecture 5:
"DbD" and data abstraction;
Introduction to Recursion

Fall 2007

Nate Titterton
nate@berkeley.edu

Schedule

3	Sep 10-14	Lecture: Conditionals, Case Studies Reading: "Difference between Dates" case study, in the reader (first version) Lab: Explore "Difference between Dates“ Start miniproject 1
4	Sep 17-21	Lecture: Data abstraction in DbD; Introduction to recursion Lab: Work on miniproject I Begin recursion Reading: Simply Scheme, Chap 11 (for Thur/Fri)
5	Sep 24-28	Lecture: Recursion Lab: More complex recursion Reading: “Dbd, recursive solution” case study “Roman Numerals” case study
6	Oct 1-4	Lecture: *Midterm 1* Lab: Advanced recursion

Announcements

- Nate's office hours (this week):
  - Wed, 2-4, 329 Soda
- Reading for this week
  - Simply Scheme, chapter 11
  - You really 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 Sept 28th
- Midterm 1 is in 2 weeks (Oct 1st)
  - It probably won't be in this room
  - 90 minutes long (4:10-5:40)
  - Open book, open notes, no computers…
  - There will be a 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

<u>Constructors</u>: procedures to make a piece of data

- `word`, `sentence`

<u>Selectors</u>: procedures to return parts of that data piece

- `first`, `butfirst`, etc.

<u>Tests</u>: 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 <test>

     (<do the base case>)

     (<do the recursive case>)


     ))
```

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
     ))
```

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? (bf sent))   ;last one?
         1                    ;base case: return 1

        ((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

   ))
```

This one has the error – if the last number in the sentence is odd, this will return a count one too large.

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 even-numbers (2)

```
(define (count-evens sent)

  (cond ((empty? (bf sent))   ;last one?
         1                    ;base case: return 1

        ((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

   ))
```

This one has the error – if the last number in the sentence is odd, this will return a count one too large.

Count the even-numbers (2)

```
(define (count-evens sent)

   (cond ((empty? (bf sent))   ;last one?
          (if (even? (bf sent))
            1 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

    ))
```

This one works, but it is ugly. Why do the check for even/odd in the base case, when the recursive cases are already doing it?

Count the even-numbers (2)

```
(define (count-evens sent)

  (cond ((empty?     sent)   ;last one?
         0                   ;base case: return 1

        ((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

   ))
```

Yeah, this one works, and looks good. The base case is simpler when it checks for the empty list, rather than the list with one left…