Lecture 7: Programming methodology and data abstraction

CS 61A

Summer 2006

Instructor: Kevin Lin

July 6, 2006

Administrative stuff

- **HW 1A, 1B due yesterday night!**
- **Project 1 due tonight!**
- **Please fill out survey if you haven't already.**
- **Midterm 1 will be on Friday, 7/14/2006, 4-6 PM, in 534 Davis.**
  - **It will emphasize weeks 1 and 2, but it may cover some week 3 material as well. If you have any sort of conflict with this time or if you need any sort of additional accommodations, please e-mail me as soon as possible so we can work something out. If you've already talked to me about this, please e-mail me again to remind me. Thanks!**

Administrative stuff

- **Reminder: Late homework assignments are NEVER accepted. You can submit project assignments up to 24 hours late, but if you do, you will get at most 50% credit on the project. Projects that are more than 24 hours late get no credit.**

Today's lecture

- Review of various topics, mostly stuff from week 1.
- More software engineering sort of stuff:
  - Programming methodology: commenting your code, naming variables, indenting, etc
  - Data abstraction, abstraction barrier

Review

What is applicative order evaluation?
What is normal order evaluation?
Which type of evaluation does Scheme use?

What is the name of the model of computation that we are currently using?

Review

How many calls to * occur when the following is evaluated with applicative order evaluation? How about with normal order evaluation?

```
(define (f x)
  (g x)
(define (g x)
  (+ x x))

(f (* 2 2))
```

Review

From the homework:
In lecture we have seen every, which takes a procedure and a sentence and applies the procedure to every word of that sentence.
Now write a procedure make-every that takes a procedure and returns another procedure that takes a sentence as argument and applies the procedure to every word of the sentence, e.g.

```
> (make-every square)
#[closure ...]
> ((make-every square) ’(1 2 3 4))
(1 4 9 16)
```

7

Review

```
(define (make-every func)
  (lambda
        (sent)
    (every func sent)))

(define (make-every func)
  (define (helper sent)
    (if (empty? sent)
        ‘()
       (se (func (first sent))
           (helper (bf sent)))))
  helper)
```

8

Review

Write a procedure `collapse` that takes a sentence of words and a predicate, and returns all of the words that satisfy that predicate collapsed into a single word.

```
> (collapse even? ‘(1 2 3 4 5 6))
246
> (collapse vowel? ‘(e x e a x o))
eeao
```

9

Review

```
(define (collapse pred sent)
  (define (helper sent)
    (if (empty? sent)
        “”
        (word (first sent) (helper (bf sent)))))
  (helper (keep pred sent)))
```

10

Review

Now write make-collapse:

```
> ((make-collapse even?) ‘(1 2 3 4 5 6))
246
> ((make-collapse vowel?) ‘(e x e a x o))
eeao
```

```
(define (make-collapse pred)
  (lambda (sent)
     (collapse pred sent)))
```

11

Programming methodology: Documentation

The secret to good documentation is to get a lot of mileage out of each comment by focusing on the things that are applicable to more than one procedure.

For example, in the twenty-one project a central concept is a particular kind of procedure called a “strategy.” So that term can be defined in a comment at the beginning of the program, along with other names for special kinds of data.

12

Programming methodology: Documentation

```
;; Data types:
;; RANK
;;  One of the following words: A2345678910JQK
;;  representing Ace, Two, Three, ... Jack, Queen, King.
;; SUIT
;;  One of the following words: HSDC
;;  representing Hearts, Spades, Diamonds, Clubs.
;; CARD
;;  A word consisting of a RANK followed by a SUIT
;;    e.g., 10H for the Ten of Hearts.
;; HAND
;;  A sentence of CARDs
;; STRATEGY
;;  A procedure that takes two arguments:
;;    a HAND (the player's hand)
;;    a CARD (the dealer's visible card)
;;  and returns a Boolean:
;;    #T if the player should take another card
;;    #F if the player should not take another card
```

13

Programming methodology: Documentation

Once these definitions are in your program, you can use the data type names as part of your formal parameters, and then you shouldn't need comments about the domain of each procedure. For example, you can write

```
(define (stop-at-17 my-hand dealer-up-card)
  ...)
```

The meaning of my-hand and dealer-up-card is clear, and no further documentation is needed.

14

Programming methodology: Documentation

You might want to note that stop-at-17 returns a boolean (recall the importance of domain and range!!!).

```
(define (stop-at-17 my-hand dealer-up-card)
  ;; returns a boolean
  ...)
```

15

Programming methodology: Documentation

However, it would be better to say that it is a strategy, which, given the previous documentation, automatically implies that it returns a boolean.

```
(define (stop-at-17 my-hand dealer-up-card)
  ;; this is a strategy procedure
  ...)
```

Alternatively, we could have called our procedure stop-at-17-strategy, except that in the project you were told what to call the procedure. Plus, "stop-at-17-strategy" might be too long to type for your preferences.
Documentation is more of an "art" than a "science"!

16

Indenting

Proper indentation can make code a lot easier to read.

```
(if something
    (if something-else
        then-this
    else-this)
    else-that)
```

Whoops! Which if statement does else-this belong to?

Some programming languages (Python) REQUIRE proper indentation.

17

Documentation and style

As we have already seen, it's good style to give names to variables that suggest the purpose of those variables.

Obviously it would be very bad to call a variable "sent" when we're expecting it to be a number.

We want readable, easy to understand code.

In future projects, we will be grading your coding style, your commenting, and your documentation. (Style will probably be worth 2-4 points out of the possible 20.) We won't be doing this for the twenty-one project.

18

Data abstraction

NOTE: We will be talking about data abstraction in the context of the twenty-one project. However, you don't have to worry about data abstraction for proj1. You'll have to worry about it later.

Philosophy: If we are dealing with some particular type of data, we want to talk about it in terms of its meaning, not in terms of how it happens to be represented in the computer.

Data abstraction

Example: Here is a function that computes the total point score of a hand of playing cards.

```
(define (total hand)
  (if (empty? hand)
      0
      (+ (butlast (last hand))
         (total (butlast hand)))))
```

Butlast appears twice … but the two instances of butlast have different meanings!

Data abstraction

```
(define (total hand)
  (if (empty? hand)
      0
      (+ (rank (one-card hand))
         (total (remaining-cards hand)))))

(define rank butlast)
(define suit last)
(define one-card last)
(define remaining-cards butlast)
```

There's more typing to do, but the program is much more readable and easier to understand. Subsequently, it's also easier to modify.

Data abstraction

The auxiliary functions like rank are called <u>selectors</u> because they select one component of a multi-part datum.

Actually, we're now violating the data abstraction if we type in a hand of cards as '(3h 10c 4d) because that assumes we know how the cards are represented. To hide the representation, we need <u>constructor</u> functions as well as the selectors:

```
(define (make-card rank suit)
  (word rank (first suit)))
(define make-hand se)
```

Data abstraction

Respecting the data abstraction barrier.

These is wrong; they contain data abstraction violations:

```
> (total '(3h 10c 4d))
> (butlast (make-card 10 'club))
```

These are right:

```
> (total (make-hand (make-card 3 'heart)
                    (make-card 10 'club)
                    (make-card 4 'diamond)))
> (rank (make-card 10 'club))
```

Data abstraction

Once we're using data abstraction we can change the implementation of the data type without affecting the programs that use that data type. This means we can change how we represent a card, for example, without rewriting total.

Data abstraction

```
(define (make-card rank suit)
  (cond ((equal? suit 'heart) rank)
        ((equal? suit 'spade) (+ rank 13))
        ((equal? suit 'diamond) (+ rank 26))
        ((equal? suit 'club) (+ rank 39))
        (else (error "say what?")) ))

(define (rank card) (remainder card 13))

(define (suit card) (nth (quotient card 13)
'(heart spade diamond club)))
```

Here, cards are represented as numbers. (Maybe we're programming on a machine that can only deal with numbers, no letters.)

Data abstraction

Advantages of data abstraction:

Code is easier to read, and often easier to write, too.

Programs are more extensible.

Can change the representation of data without changing too much code. You only need to change your <u>selectors</u> and <u>constructors</u>. All the other code can stay the same.

Compare this to Henry Ford's assembly line.