Lecture 12: Binary Search Trees, Scheme-1, … representing abstract data

July 17, 2006
CS 61A, Summer 2006
Instructor: Kevin Lin

Administrative stuff

- Midterm has not been graded yet. It should be graded by tonight.
- Project 1 grades are slowly trickling in; you'll get your papers back in section.
- Project 3 and 4 will be done in groups of two.
- Unfortunately, I think there are an odd number of people in this class.

Administrative stuff

- Bad news, which I mentioned last week:
  Due to an extremely serious family emergency, I have no choice but to go home to Southern California for the rest of the summer. I'll be around this week, and Ana Ramirez Chang will be taking over the course starting next week.
- However, I will remain involved with the course. I will be glad to answer newsgroup questions, e-mail, etc. I may hold "online office hours" if there is a demand for it. I apologize for possibly disrupting the course, but there is nothing I can do about my situation.

Today's lecture and beyond

- Some information on project 2
- Binary search trees: another application of trees (and a different implementation of trees)
- Scheme-1: an introduction to implementing an interpreter for a programming language
- The last half of today's lecture and the rest of the week: **representing abstract data** and object-oriented programming
- And beyond: Mutable data (non-functional programming), metacircular evaluator (a full Scheme interpreter), other programming paradigms and how to implement them.

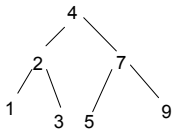
Project 2 information

- You can't really test proj2 until you've written (almost) all of the procedures. So you should be very careful with your code and with the data abstraction. DATA ABSTRACTION VIOLATIONS WILL BE PENALIZED
- Data types:
  Vectors run from the origin to a specified point (implemented as a pair with the x-coord and y-coord of the endpoint)
- Segments represent lines connecting points (implemented as pairs of vectors; the segment is the line between the endpoints of the two given vectors)
- Painters are made up of segments (implemented as a procedure that takes a frame as argument)
- Frames are parallelograms in which painters are drawn (implemented as an aggregate of three vectors)

Binary Search Trees

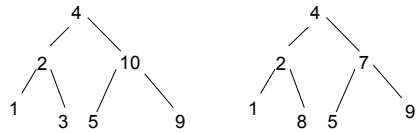
- A binary tree is a tree in which each node has at most two children. The two children are called the left-child and the right-child respectively.
- A binary search tree is a special type of binary tree containing numbers.

Binary Search Trees



This is a **binary search tree**: everything to the left of a given node is less than that node's datum; and everything to the right of a given node is greater than that node's datum.

Binary Search Trees



These are binary trees, but
they are not binary search trees. Why not?

Binary Search Trees

Implementing binary trees:

```
(define (make-binary-tree datum
          left-child right-child)
  (list datum left-child right-child))

(define (datum t)
  (car t))
(define (left-child t)
  (cadr t))
(define (right-child t)
  (caddr t))
```


Binary Search Trees

When we implemented Trees (with a capital T), we said that there was no such thing as an empty Tree.

However, we will implement empty binary trees: the empty binary tree is nil.

```
(define (empty-binary-tree? t) (null? t))
(define the-empty-binary-tree nil)
```


Binary Search Trees

```
(define (contains? x bst)
  (cond ((empty-binary-tree? bst) #f)
        ((equal? x (datum bst)) #t)
        ((> x (datum bst))
         (contains? x (right-child bst)))
        (else (contains? x (left-child bst)))))
```

Searching for elements of a binary search tree is much faster than searching for elements of a regular Tree (with a capital T)!

Scheme-1

Scheme-1 is a baby Scheme interpreter. It has lots of functionality but the main thing it is missing is DEFINE.

```
(define (scheme-1)
  (display "Scheme-1: ")
  (print (eval-1 (read)))
  (scheme-1))
```

Scheme-1

A simplified version of eval-1:

```
(define (eval-1 exp)
  (cond ((constant? exp) exp)
    ((symbol? exp) (eval exp)) ; Scheme's EVAL
    ((if-exp? exp)
     (if (eval-1 (cadr exp))
         (eval-1 (caddr exp))
         (eval-1 (cadddr exp))))
    ((lambda-exp? exp) exp)
    ((pair? exp) (apply-1 (eval-1 (car exp))
                          (map eval-1 (cdr exp))))
    (else (error "bad expr: " exp))))
```


Scheme-1

A simplified version of apply-1:

```
(define (apply-1 proc args)
  (cond
    ((procedure? proc)
       (apply proc args)) ; Scheme's APPLY
    ((lambda-exp? proc)
      (eval-1 (substitute
                (caddr proc); the body
                (cadr proc) ; params
                args        ; arg values
                '())))     ; ignore this for now
   (else (error "bad proc: " proc))))
```


Representing abstract data

A software engineering problem:
How to control the complexity of large systems with many small procedures that handle several types of data?

e.g. adding rational numbers / real numbers / complex numbers together; how to deal with multiple types?

Several methods of dealing with these sorts of problems:
-- tagged data
-- data-directed programming
-- message passing

Note: My approach to these methods in today's lecture may be different/simpler/easier than the book's approach.

Tagged data

Tagged data:

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))

(define (type-tag obj)
  (car obj))

(define (contents obj)
  (cdr obj))
```

Respect the data abstraction!

Tagged data

Suppose square1 and circle1 are previously defined shapes. We want to be able to call `(area square1)` and `(area circle1)` to get the areas of the shapes.

KEY POINT: we want to be able to use the same AREA procedure for both shapes.

```
(define pi 3.141592654)
(define (make-square side)
  (attach-tag 'square side))
(define (make-circle radius)
  (attach-tag 'circle radius))
(define (area shape)
  (cond ((eq? (type-tag shape) 'square)
         (* (contents shape) (contents shape)))
    ((eq? (type-tag shape) 'circle)
      (* pi (contents shape) (contents shape)))
(else (error "Unknown shape -- AREA"))))
```


Tagged data

```
(define (perimeter shape)
  (cond ((eq? (type-tag shape) 'square)
         (* 4 (contents shape)))
    ((eq? (type-tag shape) 'circle)
     (* 2 pi (contents shape)))
 (else (error "Unknown shape -- PERIMETER"))))
```

Data-directed programming

Suppose we want to implement a new type that will work with the AREA and PERIMETER procedures. How can we do this?

It is tedious: we will have to modify AREA and PERIMETER. If there are other procedures we want to use, we have to modify them, as well.

So let's look at some other alternatives …

19

Data-directed programming

Abelson and Sussman provide us with: PUT and GET

```
> (get 'foo 'bar)
#f
> (put 'foo 'bar 'hello)
okay
> (get 'foo 'bar)
hello
```

Our first departure from functional programming! (Why isn't this functional programming?)

20

Data-directed programming

```
(put 'square 'area (lambda (s) (* s s)))
(put 'circle 'area (lambda (r) (* pi r r)))
(put 'square 'perimeter (lambda (s) (* 4 s)))
(put 'circle 'perimeter (lambda (r) (* 2 pi r)))

(define (area shape) (operate 'area shape))
(define (perimeter shape)
  (operate 'perimeter shape))

(define (operate op obj)
  (let ((proc (get (type-tag obj) op)))
    (if proc
       (proc (contents obj))
       (error "Unknown operator for type"))))
```

21

Message passing

Another alternative is message passing. In message passing, every "object" is actually a procedure that takes a message.

```
(define (make-circle rad)
  (lambda (msg)
    (cond ((equal? msg  'area)
           (* pi rad rad))
          ((equal? msg 'perimeter)
           (* 2 pi rad))
          (else (error "bad message")))))
```

Advantages/disadvantages to message passing?

22