CS 61A MIDTERM 2 REVIEW SESSION Sunday, October 16, 2005

Exercise 0. What is your TA's name (spelled correctly!)? What is your discussion section number?

Exercise 1. What will Scheme print in response to the following expressions? If an expression produces an error message, you may just write "error"; you don't have to provide the exact text of the message. If the value of an expression is a procedure, just write "procedure"; you don't have to show the form in which Scheme prints procedures. **If the return value is a pair, then draw the corresponding box and pointer diagram.**

```
a. ((caddr '(1 + * 2)) 3 4)

b. (list? (cons 1 (cons 2 (cons nil nil))))

c. (list (list (list (cons 1 nil))))

d. (list 'one (cons 'foo (list '())))

e. (filter number? '(1 (a 2) ((b) 3) c 4 d))

f. '(a (b c d) ((e . f) g))

g. (cddadr '(a (((b)) c d e f) g))

h. (cdaadr '(a ((b . c)) (d)))

i. (if (= 2 3) '(1 2) '(3 4))

j. (map (lambda (x) (cons 'foo x)) '(1 (2) (3)))
```

Exercise 2. Write a predicate procedure `deep-car?` that takes a symbol and a deep-list (possibly including sublists to any depth) as its arguments. It should return true if and only if the symbol is the `car` of the list or of some list that's an element, or an element of an element, etc.

```
> (deep-car? 'a '(a b c))
#t
> (deep-car? 'a (b (c a) a d))
#f
> (deep-car? 'a ((x y) (z (a b) c) d))
#t
```

Fill in the blanks to complete the definition.

```
(define (deep-car? symbol lst)
  (if (pair? lst)
      (or (eq? symbol ________________________)
          (helper symbol ________________________))
      ________________________))

(define (helper symbol lsts)
  (cond ((null? lsts) ________________________)
        ((deep-car? symbol (car lsts))
         ________________________)
        (else ________________________)))
```

Exercise 3. An **athlete** contains 3 pieces of information: **name** (a word), **country** (a word), and **events** (a sentence containing the names of all the events the athlete participates in). The constructor:

```
(define (make-athlete name country events)
  (append (list name country) (cons events '())))
```

(a) Write the appropriate selectors for the athlete abstract data type.

```
(define all-athletes
  (list (make-athlete 'Kevin 'USA (se 'high-jump 'sleeping))
        (make-athlete 'Erin 'USA (se 'javelin 'logic))
        (make-athlete 'Wei 'China (se '100m-dash 'world-of-warcraft))
        (make-athlete 'Angela 'Singapore (se '100m-hurdles 'grading-midterms))
        (make-athlete 'Casey 'USA (se 'long-jump 'hacking-scheme))
        (make-athlete 'Robert 'USA (se '100m-dash 'treasure-hunting))))
```

(b) Write a procedure `(get-participants event athletes)` that returns a sentence containing the names of all the athletes which participate in that event.

```
> (get-participants '100m-dash all-athletes)
(wei robert)
```

Exercise 4. Here is the code for `eval-1`.

```
(define (eval-1 exp)
  (cond ((constant? exp) exp)
        ((symbol? exp) (eval exp))
        ((quote-exp? exp) (cadr exp))
        ((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))))
```

Suppose we moved the moved the `cond` clause beginning with `pair?` (the one that handles function calls) to the beginning of the `cond`, making it the first clause:

```
(define (eval-1 exp)
  (cond ((pair? exp) ...)
        ((constant? exp) ...)
        ((symbol? exp) ...)
        ...
        (else (error ...))))
```

With the above changes, show what Scheme-1 would print given the following inputs. If the result is an error, just write "error".

```
Scheme-1: 3
```

```
Scheme-1: +
```

```
Scheme-1: (if #t 1 2)

Scheme-1: ((lambda (x) (* x x)) 3)

Scheme-1: (+ 3 4)

Scheme-1: (lambda (x) (* x x))
```

Exercise 5. Write the function `datum-filter` which, given a predicate and a tree, returns a list of all the data (that's plural for datum!) which satisfy the predicate (in any order). Do this for general trees, not binary trees. The function should return the empty list for any tree in which no data satisfy the predicate. You may use helper procedures.

Exercise 6. Write `double-datum?`, which takes a tree as its argument, and returns true if there exists a node anywhere in the tree that has the same datum as one of its children (immediate children, not grandchildren or cousins), otherwise false.

Exercise 7. Instead of attaching one tag to an object, suppose we wanted to attach multiple tags. After all, a tomato is not only a food, it's also a projectile! So we'll rename `attach-tag` to `attach-tags` (taking a list of tags as its first argument), and rename `type-tag` to `type-tags`.

Here is the original version of `operate`:

```
(define (operate op obj)
  (let ((proc (get (type-tag obj) op)))
    (if proc
        (proc (contents obj))
        (error "No such operator for this type"))))
```

Rewrite `operate` so that it looks for each of an object's type tags in the table, using the first one for which a procedure is found for this operator.

Exercise 8. Write a procedure `(flatten ls)` that takes in a deep list and flattens all of the elements into a flat list.

```
> (flatten '((1) (2 ((3 4) ((5 6) 7) 8) 9) 10))
(1 2 3 4 5 6 7 8 9 10)
> (flatten '(1 2 3 4))
(1 2 3 4)
> (flatten '((((((1)))))))
(1)
```

Exercise 9. Below are definitions of `m-car` and `m-cdr` (the `m` here stands for "message-passing").

```
(define (m-car pair) (pair 'car))
(define (m-cdr pair) (pair 'cdr))
```

(a) Write a definition of `m-cons` that behaves just like regular `cons` when used in conjunction with `m-car` and `m-cdr`.

(b) If you type `list1` at the STk prompt, what would it print out?

```
(define list1 (m-cons 1 (m-cons 2 (m-cons 3 '()))))
```

(c) We would like to be able to convert a message-passing list into an ordinary Scheme list. Write the function `m-list-to-regular-list` that takes a message-passing list (hmm, I haven't yet defined what a message-passing list is; how should it be defined?) and returns a regular list of the same elements.

Exercise 10. Here is the code for `apply-1`.

```
(define (apply-1 proc args)
  (cond ((procedure? proc)
         (apply proc args))
        ((lambda-exp? proc)
         (eval-1 (substitute (caddr proc)  ; the body
                             (cadr proc)   ; the formal parameters
                             args          ; the actual arguments
                             '())))        ; bound-vars
        (else (error "bad proc: " proc))))
```

Suppose we changed it so that we no longer called `substitute` inside of `apply-1`:

```
(define (apply-1 proc args)
  (cond ((procedure? proc)
         (apply proc args))
        ((lambda-exp? proc)
         (eval-1 (caddr proc)))  ; evaluate the body without using substitute
        (else (error "bad proc: " proc))))
```

With the above changes, show what Scheme-1 would print given the following inputs. If the result is an error, just write "error".

```
Scheme-1: ((lambda (x) (* x x)) 3)

Scheme-1: ((lambda (x) (+ 2 3)) 2)

Scheme-1: (+ 3 4)

Scheme-1: (lambda (x) (* x x))

Scheme-1: (lambda (x) (+ 3 4))

Scheme-1: ((lambda (cons) (cons 1 2)) +)
```

Exercise 11. Write a function `max-children` that returns the maximum number of children of any node in the tree. For example, suppose the following tree is `mytree`:

```
      1
     /|\
    2 3 4
   / \   \
  5   6   7
 /|\
8 9 10
```

In this case, `(max-children mytree)` would return 3.