CS3 Midterm 2

Summer 2008

Read this page and fill in the left table now.

Solutions

login (eg, cs3-ab):	
UCWISE login:	
Name of the person sitting to your **left**:	
Name of the person sitting to your **right**:	

You have 120 minutes to finish this test, which should be reasonable. Your exam should contain 7 problems (numbered 0-7) on 11 total pages.

This is an open-book test. You may consult any books, notes, or other paper-based inanimate objects available to you. Read the problems carefully. If you find it hard to understand a problem, ask us to explain it.

Restrict yourself to Scheme constructs covered in this course. (Basically, this excludes chapters 16 and up in Simply Scheme).

Please write your answers in the spaces provided in the test; if you need to use the back of a page make sure to clearly tell us so on the front of the page.

Partial credit will be awarded where we can, so do try to answer each question.

Good Luck!

(1 pt)	Prob 0	
(6)	Prob 1	
(6)	Prob 2	
(7)	Prob 3	
(3)	Prob 4	
(6)	Prob 5	
(10)	Prob 6	
(5)	Prob 7	
Raw Total (out of 44)		
Scaled Total (36)		

Problem 0. (1 point)

Put your login name on the top of each page.
Also make sure you have provided the information requested on the first page.

Problem 1. Many ways to end a word (A: 1 point; B: 2 point; C: 1 point; D: 2 points)

The problems below involve adding a suffix to a word. For example adding 'ing to the word `'eat` to make `'eating`.

Part A: (1 point) Based upon the definition of `suffix-adder1` below, write a call to `suffix-adder1` to return the value of adding the suffix `'ing` to the word `'eat`.

```
(define (suffix-adder1 suffix)
  (lambda (wd) (word wd suffix)))
```

((suffix-adder1 'ing) 'eat)

Notes: (suffix-adder1 'ing) returns a procedure that takes in one argument. So we call the procedure that this returns with the argument 'eat.

Part B: (2 point) Using the definition of `suffix-adder1` below, write the procedure `add-ings1` using higher order procedures (and no recursion). `add-ings1` that should take a sentence as an argument and add `'ing` to every word in the sentence.

- You may NOT use recursion
- You may use higher order procedures
- You MUST use the definition of `suffix-adder1` provided

```
(define (suffix-adder1 suffix)
  (lambda (wd) (word wd suffix)))
```

(define (add-ings1 sent)
(every (suffix-adder1 'ing) sent))

Notes: We want to apply the procedure that suffix-adder1 returns to every word in the sentence. This works because (suffix-adder1 'ing) returns a procedure that takes in one argument. (So it is compatible with every).

Question 1 (continued)

Part C: (1 point) Based upon the definition of `suffix-adder2` below, write a call to `suffix-adder2` to return the value of adding the suffix `'ing` to the word `'eat`.

```
(define suffix-adder2 (lambda (wd suffix) (word wd suffix)))
```

```
(suffix-adder2 'eat 'ing)
```

Notes: The definition of suffix-adder2 above is the same as if you had said:

```
(define (suffix-adder2 wd suffix)
     (word wd suffix))
```

So this is just like a normal procedure that takes in two arguments.

Part D: (2 point) Using the definition of `suffix-adder2` below, write the procedure `add-ings2` using higher order procedures (and no recursion). `add-ings2` that should take a sentence as an argument and add 'ing to every word in the sentence.

- You may NOT use recursion
- You may use higher order procedures
- You MUST use the definition of `suffix-adder2` provided

```
(define suffix-adder2 (lambda (wd suffix) (word wd suffix)))
```

```
(define (add-ings2 sent)
  (every
    (lambda (wd) (suffix-adder2 wd 'ing))
     sent))
```

Notes: The procedure suffix-adder2 takes in two arguments, so we can't just use it automatically with every like in part B. We need to use lambda so that we have access to the two variables. Our procedures is still going to be suffix-adder2, but we're going to use lambda to make sure it has access to both each word in the sentence and 'ing.

Problem 2. Plus-One (6 points)

Using only recursion and no helper procedures, write the procedure `plus-one` that takes a sentence representing a number and adds one to the number that the sentence represents. The sentence will contain zero or more words, where each word is a number between 0 and 9.

```
(plus-one '( 5 )) -> '( 6 )
(plus-one '( 2 3 4 )) -> '( 2 3 5 )
(plus-one '( 9 9 9 )) -> '( 1 0 0 0 )
```

- Use recursion
- Do not use higher order procedures
- Do not use helper procedures

```
(define (plus-one sent)
     (cond ((empty? sent)
               (se '1))

          ((> 9 (last sent))
               (se (butlast sent) (+ 1 (last sent))))

          (else
               (se (plus-one (butlast sent)) '0)))
```

Notes: The easiest case is the second one. If the number is 0-8 we should just add one to it and we're done with the recursion. The third case handles when the number IS 9. In this case, we should replace the 9 with a zero and carry the one to the next digit by calling plus-one on the butlast of the sentence. Given that we keep removing the last number in the sentence, carrying one is functionally the same as adding one.

Problem 3. Is that a sentence over there? (7 points)

Someone has gotten confused between the difference between words and sentences! They have written a sentence as a scheme word, with each word in the sentence separated by a single asterisk (*). For example these are all sentences that they wrote:

`'this*is*a*sentence`
`'a*crazy*sentence`
`'cs3*totally*rocks`
`'hello`

Write a procedure `split-word` that takes in a word of this form and returns a sentence. **You may NOT use recursion. You should use higher order functions.**

`(split-word 'this*is*a*sentence)`	➔	`'(this is a sentence)`
`(split-word 'a*crazy*sentence)`	➔	`'(a crazy sentence)`
`(split-word 'cs3*totally*rocks)`	➔	`'(cs3 totally rocks)`
`(split-word 'hello)`	➔	`'(hello)`

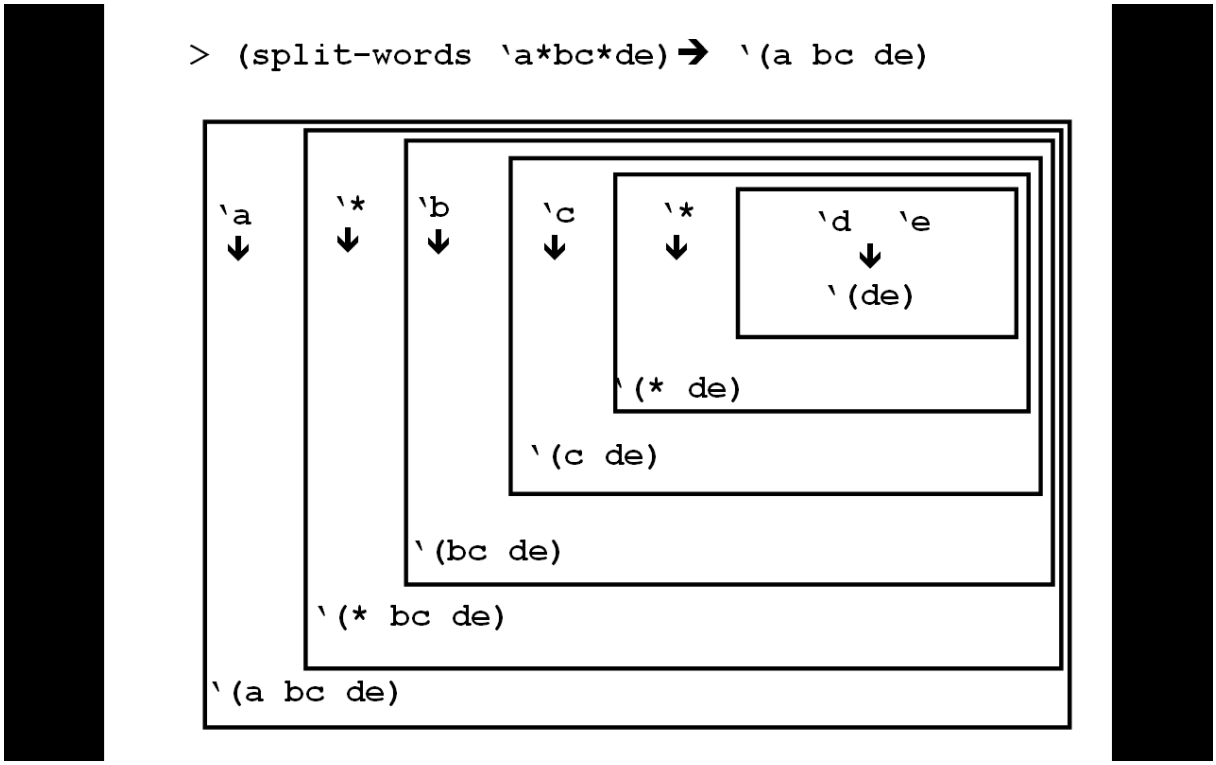
5 points for the procedure `split-word`

2 points for comments explaining how the procedure works

(there is more room on the next page)

(continued) Problem 3: Is that a sentence over there?

```
(define (split-words wd)
  (accumulate
    (lambda (char so-far)
      (cond
       ((equal? char '*)
            (se char so-far))
       ((equal? (first so-far) '*)
            (se char (bf so-far)))
       ((empty? (bf so-far))
            (se (word char (first so-far))))
       (else
            (se
               (word char (first so-far))
               (bf so-far)))))
    wd))
```



Notes:

We're going to use accumulate to break up each letter of the word and combine it into one finished sentence. The "so-far"

argument of the lambda is going to maintain the sentence we've constructed so-far.

The easiest case to understand is the else case (4th case). If we're in the middle of one word we might have char equal 't and so-far equal '(otally rocks). In the else case, we add the char to the first word in the so-far.

The first time the lambda procedure is called it will be passed two characters. So in the third case, if the bf of the so-far is empty, then this is the first time the lambda procedure has been called. If it is the first time the procedure has been called then we should put those two letters together in a word and then put that word in a sentence.

When we find a '* we know that the next character should start a new word. So if we find a '* then we're going to sentence that '* to the front of the so-far (case 1).

In case two, we find that the first of the so-far is a '*. This means that we need to replace that '* and start a new word.

Problem 4. More prefixes! (3 points)

We want to modify how roman numerals work so that you can have multiple prefixes. For example, the new version of roman-sum should result in continually subtracting any prefix from the next number.

```
(roman-sum '(1 5 10)) ➔ 6
(roman-sum '(1000 1 5 10)) ➔ 1006
(roman-sum '(1 10 100)) ➔ 91
(roman-sum '(1000 1 10 100)) ➔ 1091
```

(- 10 (- 5 1))

Fill in the code below to modify the initial version of `roman-sum`. You only need to modify the case when the `number-sent` starts with a prefix. (A copy of the original code is available in the appendix..)

```
(define (roman-sum number-sent)
 (cond
   ((empty? number-sent) 0)
   ((empty? (bf number-sent)) (first number-sent))
   ((not (starts-with-prefix? number-sent))
    (+ (first number-sent) (roman-sum (bf number-sent)) ) )
   ((starts-with-prefix? number-sent)
```

```
(roman-sum
   (se
       (-
          (first (bf number-sent))
          (first number-sent))
      (bf (bf number-sent))))))
```

Problem 5. Predict the output (10 points)

Write the result of evaluating the Scheme expression that comes before the ➔. If the Scheme expression will result in an error, write *ERROR* in the blank and describe the error.

1

```
(count (accumulate word '(a b c d e f)))➔ 6
```

2

```
(keep (equal? 'b) '(a b c b d)) ➔ error. (equal? 'b)
                                   is not a procedure
```

3

```
(define (add-k sent)
  (if (empty? sent)
      sent
      (sentence
               (word (first sent) 'k)
               (add-k (first (butfirst sent))))))
```

```
(add-k '(cat dog hat bat)) ➔ error. Invalid arg to
                                    First
```

4

```
(accumulate - '(10 3 4)) ➔ 11
```

5

```
(define (mystery num)
   (if (= num 0)
       '()
       (sentence
                num
                (mystery (- num 2))
                (mystery (- num 2)))))
```

```
(mystery 6)) ➔  '(6 4 2 2 4 2 2)
```

```
(mystery 3)) ➔  error. Infinite loop
```

6

```
(define (odds-please sent)
   (cond ((empty? sent)
                #f)
         ((odd? (first sent))
                (se(first sent) (odds-please (bf sent))))
         (else (odds-please
                (bf sent)))))
```

(odds-please '(2 4 6 8)) ➔ **#f**

(odds-please '(2 3 5 8)) ➔ **error. You can't sentence #f because it is not a word**

7 ((lambda (x) (* x x)) 3) ➔ **9**

8 (every odd? '(1 2 3 4 5)) ➔ **error. Every only takes procedures that return a word or a sentence because you cant sentence together #t and #f**

Problem 6. Reverse Pairs (5 points)

Write the procedure `reverse-pairs` that takes in a sentence and returns a sentence with adjacent pairs switched. If there are an odd number of words in the sentence the last word in the sentence should not be switched with any other word.

For example
`(reverse-pairs '(a b c d e f))` ➔ `'(b a d c f e)`

`(reverse-pairs '(a))` ➔ `'(a)`

`(reverse-pairs '(a b c))` ➔ `'(b a c)`

```
(define (reverse-pairs sent)
    (cond
        ((empty? sent) sent)
        ((= 1 (count sent)) sent)
        (else (se
            (first (butfirst sent))
            (first sent)
            (reverse-pairs (bf (bf sent)))))))
```

Notes: The important part here is to make sure that you have both a first and a second before you try to take the (bf (bf sent)) or try to rearrange the first two elements.

Roman Numeral Case Study
(Appendix A with rewritten functions in appendix B, in the reader)

```
; Return the decimal value of the Roman numeral whose digits are
; contained in roman-numeral.
; Roman-numeral is assumed to contain only Roman digits.
; Sample call: (decimal-value 'xiv), which should return 14.
(define (decimal-value roman-numeral)
  (roman-sum
    (digit-values roman-numeral) ) )


; Return a sentence containing the decimal values of the Roman digits
; in roman-numeral.
; Roman-numeral is assumed to contain only Roman digits.
; Sample call: (digit-values 'xiv), which should return (10 1 5).
(define (digit-values roman-numeral)
  (if (empty? roman-numeral) '( )
    (se  (decimal-digit-value (first roman-numeral))
         (digit-values (bf roman-numeral)) ) ) )

; Return the decimal value of the given Roman digit.
(define (decimal-digit-value roman-digit)
  (cond
    ((equal? roman-digit 'm) 1000)
    ((equal? roman-digit 'd)  500)
    ((equal? roman-digit 'c)  100)
    ((equal? roman-digit 'l)   50)
    ((equal? roman-digit 'x)   10)
    ((equal? roman-digit 'v)    5)
    ((equal? roman-digit 'i)    1) ) )


; Return the decimal value of a Roman numeral. The decimal equivalents
; of its Roman digits are contained in number-sent.
; Sample call: (roman-sum '(10 1 5)), which should return 14.
(define (roman-sum number-sent)
  (cond
    ((empty? number-sent) 0)
    ((empty? (bf number-sent)) (first number-sent))
    ((not (starts-with-prefix? number-sent))
     (+ (first number-sent) (roman-sum (bf number-sent)) ) )
    ((starts-with-prefix? number-sent)
     (+
        (- (first (bf number-sent)) (first number-sent))
        (roman-sum (bf (bf number-sent))) ) ) ) )


; Return true if the number-sent starts with a prefix, i.e. a number
; that's less than the second value in the sentence.
; Number-sent is assumed to be of length at least 2 and to contain
; only positive numbers.
(define (starts-with-prefix? number-sent)
  (< (first number-sent) (first (bf number-sent))) )
```