Read and fill in this page now

Your name: _____

Your instructional login (e.g., cs3-ab): _____

Your lab section days and time: _____

Your lab T.A.: _____

Name of the person sitting to your left: _____

Name of the person sitting to your right: _____

Prob 0:		Prob 1:		Prob 2:	
Prob 3:		Prob 4 (a/b/c):			
---	---	Prob 5 (a/b/c):			
Raw Total:	/50		**Scaled Total**:		/30

You have 70 minutes to finish this test, which should be reasonable; there will be approximately 10 additional minutes of leeway given. Your exam should contain 6 problems (numbered 0-5) on 10 total pages. It includes the code from the case study *Difference Between Dates Part II* in an appendix at the end.

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. If you have a question during the test, please come to the front or the side of the room to ask it.

Restrict yourself to Scheme constructs covered in chapters 3-6 and 11 of *Simply Scheme*, the *Difference Between Dates* case study, parts 1 and 2, and the *Roman Numerals* case study. You can always use helper procedures, and procedures from other questions you've answered.

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. We believe we have provided more than enough space for your answers, so please don't try to fill it all up.

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

Relax!

Problem (1 point, 1 minute)

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

Problem (7 points, 11 minutes). Fill in the blanks.

Each of the following parts has a blank that you need to fill in. For parts *(1)-(5)*, the blank follows an ➔; fill in 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.

	`(word 'go (word 'cal (word 'for (word 'evah (word ""))))) ➔ _____`
	`(se 'go (se 'cal (se 'for (se 'evah (se ""))))) ➔ _____`
	`(word (sentence 'word 'is 'the 'word)) ➔ _____`
	`(sentence (word 'word 'is 'the 'word)) ➔ _____`
	`(+ 3 _____ 5 ) ➔ ERROR`

For parts *(6)-(8)*, fill in the blank so that the resulting scheme expression evaluates to the result shown. Don't use any parentheses. If it is impossible to do so, write *IMPOSSIBLE* in the blank.

	`(define (doit x) (cond ((equal? x 'a) 'hah) ((equal? x 'b) 'hab)))` `(doit _____ ) ➔ ERROR`
	`(or _____ 'joe 'bob) ➔ 'joe`
	`(and _____ 'joe 'bob) ➔ 'joe`

Problem (7 points, 10 minutes). Line-em up and add-em.

Write a procedure `add-em` which takes a sentence as input, and returns a number. The result number should be the sum of the numbers in the input sentence, starting at the beginning and continuing until something other than a positive number is reached.

You may use helper procedures.

```
                (add-em '(1 4 2 0 934 -3 5))  ➔ 7
                     (add-em '(3 5 a 8 j 2))  ➔ 8
                         (add-em '(1 2 3 4))  ➔ 10
(add-em '(fred and sally sitting in a tree))  ➔ 0
```

Problem (8 points, 12 minutes): Celebrity poker needs programmers like you.

Write `card-outranks?` The procedure takes two cards and returns true if and only if the first card is bigger than second.

Cards are represented by a two-character word, where the first character represents the rank (`a`, `k`, `q`, `j`, `0`, `9`, `8`, `7`, `6`, `5`, `4`, `3`, and `2`), and the second character represents the suit (`s`, `h`, `d`, and `c`). For instance, `2h` is the two of hearts, `qc` is queen of clubs, `0s` is the 10 of spades, etc. For this problem, consider *all* spades to rank higher than hearts, which all rank higher than diamonds, which all rank higher than clubs.

```
(card-outranks? 'ac '3d) ➔ #f
(card-outranks? 'kh 'qh) ➔ #t
(card-outranks? '4s '4s) ➔ #f
```

Comment all your procedures. Assume you have a working version of `outranks?`, as you wrote in lab, to use. (Remember, `outranks?` takes two ranks and returns true if the first is higher than the second.)

You need to use proper abstraction. In this case, you will need to define accessors, name them meaningfully, and include comments indicating their purpose.

Problem (3/3/12 points, 24 minutes): Can you span this?

Part a (3 points). Write `days-until-new-year`, which takes a date and returns the number of days until the end of the year, inclusive.

Remember that you have the procedures in the *Difference between Dates case study* at your disposal, including the `day-span` procedure. The answer should fit in the space below!

```
  (days-until-new-year '(january 3)) ➔ 363
(days-until-new-year '(december 31)) ➔ 1
```

Part b (3 points). Write `hours-until-new-year`, which takes a date and returns the number of hours until the end of the year. (Assume that you need to calculate from noon of the date given).

```
  (hours-until-new-year '(january 3)) ➔ 8700
(hours-until-new-year '(december 29)) ➔ 60
(hours-until-new-year '(december 31)) ➔ 12
```

Part c (12 points). The following are buggy versions of the recursive procedure `day-sum`, defined in the cases study *Difference between dates, part II.* (The code for the case study is included as an appendix). The bugs result from small changes which are underlined.

For each version, note whether the bug creates a problem in the

- conditional ,
- the base case,
- making the problem smaller,
- calling the function recursively, or
- combining the recursive calls.

Also briefly describe in English the effect of the bug on the operation of `day-span` as a whole (*not just on `day-sum`*)—this should take between 1 and 2 sentences for each case. You might include an example call to day-span illustrating the problem, although this isn't necessary with a sufficient explanation (and, might be wrong!).

Don't be too verbose! We may deduct points if our eyes start to bleed.

```
(define (day-sum first-month last-month)
  (if (>= first-month last-month)
    0
    (+ (days-in-month (name-of first-month))
       (day-sum (+ first-month 1) last-month))))
```

```
(define (day-sum first-month last-month)
  (if (> first-month last-month)
    0
    (+ (days-in-month (name-of first-month))
       (day-sum first-month (+ last-month 1)))))
```

Part c continued.

```
(define (day-sum first-month last-month)
  (if (<= first-month last-month)
    0
    (+ (days-in-month (name-of first-month))
       (day-sum (+ first-month 1) last-month))))
```

```
(define (day-sum first-month last-month)
  (if (> first-month last-month)
    1
    (+ (days-in-month (name-of first-month))
       (day-sum (+ first-month 1) last-month))))
```

Problem (2/3/4 points, 12 minutes): Coins.

For the following problems, you will be working with coins:

Coin	Coin name	Value (in cents)
`p`	Penny	1
`n`	Nickle	5
`d`	Dime	10
`q`	Quarter	25

Part a. Fill in the blanks to write `valid-coin?`, which takes a coin and returns `#t` if it is valid (i.e., in the table above) or `#f` otherwise:

```
(define (valid-coin? coin)

   (_________________  coin  ___________________))
```

Part b. Write `use-coin-to-pay`, which takes a numeric amount that is owed and returns the smallest-valued coin that is enough to pay that amount. If there is no single coin that will cover the entire amount to pay, return `"too-expensive"`. You can assume that `amount` will be a positive number.

```
(define (use-coin-to-pay amount)
```

Part c. Write test cases for your procedure `use-coin-to-pay` (you can abbreviate it). Make sure to include the expected return value. Include enough cases to thoroughly test your procedure.

Appendix A: *Difference Between Dates, part II* **code (given in appendix C).**

(This is not a question!)

```
; Recursive computation of the difference between dates

; Return the number of days spanned by earlier-date and later-date.
; Earlier-date and later-date both represent dates in 2002,
; with earlier-date being the earlier of the two.
(define (day-span earlier-date later-date)
  (cond
    ((same-month? earlier-date later-date)
     (same-month-span earlier-date later-date) )
    ((consecutive-months? earlier-date later-date)
     (consec-months-span earlier-date later-date) )
    (else
     (general-day-span earlier-date later-date) ) ) )

; Access functions for the components of a date.
(define (month-name date) (first date))
(define (date-in-month date) (first (butfirst date)))

; Return true if date1 and date2 are dates in the same month, and
; false otherwise. Date1 and date2 both represent dates in 2002.
(define (same-month? date1 date2)
  (equal? (month-name date1) (month-name date2)))

; Return the number of the month with the given name.
(define (month-number month-name)
  (cond
    ((equal? month-name 'january) 1)
    ((equal? month-name 'february) 2)
    ((equal? month-name 'march) 3)
    ((equal? month-name 'april) 4)
    ((equal? month-name 'may) 5)
    ((equal? month-name 'june) 6)
    ((equal? month-name 'july) 7)
    ((equal? month-name 'august) 8)
    ((equal? month-name 'september) 9)
    ((equal? month-name 'october) 10)
    ((equal? month-name 'november) 11)
    ((equal? month-name 'december) 12) ) )

; Return true if date1 is in the month that immediately precedes the
; month date2 is in, and false otherwise.
; Date1 and date2 both represent dates in 2002.
(define (consecutive-months? date1 date2)
  (=
    (month-number (month-name date2))
    (+ 1 (month-number (month-name date1))) ) )

; Return the difference in days between earlier-date and later-date,
; which both represent dates in the same month of 2002.
(define (same-month-span earlier-date later-date)
  (+ 1
    (- (date-in-month later-date) (date-in-month earlier-date)) ) )
```

```
; Return the number of days in the month named month-name.
(define (days-in-month month-name)
  (cond
    ((equal? month-name 'january) 31)
    ((equal? month-name 'february) 28)
    ((equal? month-name 'march) 31)
    ((equal? month-name 'april) 30)
    ((equal? month-name 'may) 31)
    ((equal? month-name 'june) 30)
    ((equal? month-name 'july) 31)
    ((equal? month-name 'august) 31)
    ((equal? month-name 'september) 30)
    ((equal? month-name 'october) 31)
    ((equal? month-name 'november) 30)
    ((equal? month-name 'december) 31) ) )

; Return the number of days remaining in the month of the given date,
; including the current day. Date represents a date in 2002.
(define (days-remaining date)
  (+ 1 (- (days-in-month (month-name date)) (date-in-month date))) )

; Return the difference in days between earlier-date and later-date,
; which represent dates in consecutive months of 2002.
(define (consec-months-span earlier-date later-date)
  (+ (days-remaining earlier-date) (date-in-month later-date)) )

; Return the name of the month with the given number.
; 1 means January, 2 means February, and so on.
(define (name-of month-number)
  (item
    month-number
    '(january february march april may june
      july august september october november december) ) )

; Return the sum of days in the months represented by the range
;    first-month through last-month.
; first-month and last-month are integers; 1 represents January,
; 2 February, and so on.
; This procedure uses recursion.
(define (day-sum first-month last-month)
  (if (> first-month last-month) 0
    (+
      (days-in-month (name-of first-month))
      (day-sum (+ first-month 1) last-month)) ) )

; Return the number of the month that immediately precedes the month
; of the given date.  1 represents January, 2 February, and so on.
(define (prev-month-number date)
  (- (month-number (month-name date)) 1) )

; Return the number of the month that immediately follows the month
; of the given date.  1 represents January, 2 February, and so on.
(define (next-month-number date)
  (+ (month-number (month-name date)) 1) )

; Return the difference in days between earlier-date and later-date,
; which represent dates neither in the same month nor in consecutive months.
(define (general-day-span earlier-date later-date)
  (+  (days-remaining earlier-date)
      (day-sum
        (next-month-number earlier-date)
        (prev-month-number later-date) )
      (date-in-month later-date) ) )
```