CS3 Fall 05 – Midterm 1
Standards and Solutions

Problem 1 (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.

These were worth 1 point each, and the question was capped at 7 points (so, you could miss one part and still get a full score for the question).

	`(word 'go (word 'cal (word 'for (word 'evah (word ""))))))` ➔ `gocalforevah`
	`(se 'go (se 'cal (se 'for (se 'evah (se ""))))))` ➔ `(go cal for evah "")` Missing the empty word at the end (`""`) lost you a point – this is an important issue that arises when dealing with proper base cases in a recursive procedure.
	`(word (sentence 'word 'is 'the 'word))` ➔ *ERROR* The constructor for a word can take words only, not sentences!
	`(sentence (word 'word 'is 'the 'word))` ➔ `( wordistheword )`
	`(+ 3 'fred 5 )` ➔ *ERROR* Any quoted word would work here. We also accepted a function name. If you put a symbol here without the quote, (i.e., simply `fred`) you only got ½ point, because this wouldn't be an error in situations where that symbol was defined to be a number! For instance, `(+ 3 x 5)` would evaluate to `9` inside a procedure where the parameter `x` has been set to `1`. In situations where `x` was undefined, you would get the `"undefined symbol"` error, of course.

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 'fred 'john )` ➔ *ERROR* Any two quoted words would work. We accepted `IMPOSSIBLE` here, since it might not have occurred to you to put two values inside one blank (certain TAs rebelled against the possible solution above). The error with two values would be `"too many arguments"`. But, with one argument, it is pretty much impossible to break a `cond`. Passing in `'c`, or any quoted word, would simply return `OKAY`, in STk. As with part (5), if you put an unquoted symbol here, you lost ½ a point, for the same reasoning as in (5)
	`(or #f 'joe 'bob)` ➔ `'joe`
	`(and` *`IMPOSSIBLE`* `'joe 'bob)` ➔ `'joe`

Problem 2 (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
```

Here's a solution:

```
(define (add-em sent)
   (if (or (empty? sent)
           (not (number? (first sent)))
           (not (< 0 (first sent))))
       0
       (+ (first sent) (add-em (bf sent)))))
```

Or how about this one:

```
(define (add-em sent)
   (cond ((empty? sent) 0)
         ((not (number-greater-than-zero? (first sent))) 0)
         (else (+ (first sent)
                  (add-em (bf sent))))))

(define (number-greater-than-zero? wd)
   (and (number? wd)
        (> wd 0)))
```

Basically, the tricky part of this recursive procedure was designing the conditional so that it went to the base case at the right times. There were three different situations that you had to test for, and the one that many of you forgot was the empty sentence. The third example — `(add-em '(1 2 3 4))` — wouldn't ever stop otherwise!

You recevied one point each for

- having a conditional, and getting it set up right
- the three parts of the test in the conditional
- the base case (e.g., `0`)
- the combiner `+` and `(first sent)`
- the recursive call of `(add-em (bf sent))`

Other typical problems included getting the direction of the comparison wrong (i.e., testing for > rather than <) and not stopping the recursion when a non-number or negative number was found (i.e., adding every positive number in the sentence).

Problem 3 (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.

There were a variety of ways you could go about this problem. We were strict about requiring you to use accessors for the suit and rank of a card, but were not as strict about strange constructions of the conditional. Here is a nice solution:

```
;; Returns true if the second card outrans the first
(define (card-outranks? card1 card2)
   (if (equal? (suit card1) (suit card2))
       (outranks? (rank card1) (rank card2))
       (outsuits? (suit card1) (suit card2)))

;; Accessor: gets the rank of the given card
(define (rank card)
   (first card))

;; Accessor: gets the suit of the given card
(define (suit card)
   (first (bf card)))

;; Returns true if the first suit beats the second
(define (outsuits? suit1 suit2)
   (> (suitrank suit1) (suitrank suit2))

;; Returns the rank of the suit
 (define (suitrank suit)
   (cond ((equal? suit 's) 4)
         ((equal? suit 'h) 3)
         ((equal? suit 'd) 2)
         ((equal? suit 'c) 1)))
```

You got up to 4 points for data abstraction
- +2 points for defining accessors (rank, suit)
- +2 points for commenting accessors
- -1 point for not using the defined accessors
- -1 point each for each non-meaningful accesssor names

You got up to 4 points for correctness of the card-outranks? procedure:
- -2 points if you switched the order of checking but other than that the program would have worked- i.e., doing something like:

```
   ((equal? (rank card1) (rank card2))
    (> (suit-value (suit card1)) (suit-value (suit card2))))
```

- -2 points if outranks? was called incorrectly, like (outranks? card1 card2)
- -1 point if you switched the importance of the suits, like heart>spade>club>diamond.
- up to -2 points if there were syntax errors (missing parenthesis), unbound variable errors, depending on severity of the error.

Problem 4 (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
```

We wanted you to do this and the next procedure simply, using only the `day-span` procedure from the case studies. And, the vast majority of you did so. Some solutions included

```
(define (days-until-new-year date)
   (- 366 (day-span '(january 1) date)))
```

and the simpler second line of

```
   (day-span date '(december 31)))
```

or

```
   (- 366 (day-of-year date))
```

You lost one point for a variety of mistakes, including forgetting parentheses, using 365 rather than 366, and so forth.

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
```

The last test case above gives the biggest help:

```
(define (hours-until-new-year date)
   (- (* 24 (days-until-new-year date)) 12))
```

Other solutions had second lines:

```
   (+ (* (- (days-until-new-year date) 1) 24) 12)

   (* (- (days-until-new-year date) 0.5) 24)
```

You lost one point for forgetting to deal with the 12 hours somehow.

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.

Each part was worth 3 points: 1 point for noting where the problem was, and 2 points for your description. In general, too low level of a description lost you 1 or maybe 2 points. We didn't want an english description of how the procedure worked, but a description of what its implications were.

```
(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))))
```

Conditional. [Note: this bug changes the test in the conditional, it doesn't change the value of the base case!]

The bug will cause day-sum to drop out of the recursion before it adds in the number of days in the last month. This will cause day-span to return a too-small number when called with non-equal or consecutive months.

Some of you wrote that this would add an extra month, or wrote that it would lose a middle month rather than the 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)))))
```

Making the problem smaller. [The recursive call is made in such a way that the problem doesn't get simpler each time]

The bug will cause day-sum to loop infinitely, and cause calls to day-span to never return a value (until scheme crashes). Those of you that said, simply, day-span will return an error only received one point for your explanation.

Some of you simply said that day-sum would cause an error, which wasn't enough to get full credit here. If you emphasized that "all of scheme would crash", we did you full points, but the much preferable answer included some notion of a never-ending situation.

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))))
```

Conditional.

With normal arguments to day-span (where the month of the earlier date is not later than the month of the later date), day-sum will return 0 every time. This will cause day-span to fail to add any months between dates with non-consecutive or equal months (essentially, treating these situations as if the months were consecutive).

Some of you noted that in situations where the first month was greater than the last month this would recurse forever, which was OK but not necessary (actually, the `item` call in `(name-of first-month)` will cause an error when first-month grows past 12).

```
(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))))
```

Base case. [The value that day-sum returns for the base case is wrong.]

Day-sum will return a number 1 greater than it should, because of the erroneous base case. This will cause day-span to return a number 1 greater than it should for dates that are non-equal or consecutive.

Some of you wrote that this buggy version would return extra day for each month, which is wrong.

Problem 5 (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)

   (   member?   coin     '(p n d q)   ))
```

One point for each blank.

Some of you wrote (member? coin 'pndq), which also received full credit. However, keep in mind that if someone used 'hd for half-dollar coins, then valid-coin? would crash instead of returning #f.

Some wrote (member? coin ('p 'n 'd 'q)), and got one point off. Scheme would generate an error for bad eval function because ('p 'n 'd 'q). It would try to evaluate 'p as a procedure, which it isn't, and thus crashes.

Two points were deducted for this solution: (equal? coin (or 'p 'n 'd 'q)). The function or returns the first true value, and would always return p in this case. This incorrectly returns #f for n, d, and q.

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)
```

There are several different solutions that could work here.

```
(define (use-coin-to-pay amount)
  (cond ((> amount 25) 'too-expensive)
        ((> amount 10) 'q)
        ((> amount 5) 'd)
        ((> amount 1) 'n)
        (else 'p)))
```

This part was worth 3 points. One point was deducted in the following cases:
- If the too-expensive case was forgotten or handled incorrectly.
- If the boundary cases were handled incorrectly, such as when 5 would return p or d.
- If the comparisons were backwards, ie. less than instead of greater than, or vice versa.
- If p, n, d, and q were not quoted. We have emphasized the usage of quotes many times in lab.
- If syntax and the structure of cond was incorrect, such as missing a set of parenthesis.

Some of you returned too-expensive as a sentence rather than a word. We did not take off any point for this.

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.

We were looking for three main classes of test cases. First, a test for an amount that was too large and should result in `too-expensive` being returned. Any number above 25 worked here, and you got 1 point for this test.

```
(use-coin-to-pay 35)    ;; should return 'too-expensive
```

Second, tests that looked at the boundary cases where amount was exactly equal to a coin value. These would test whether our > or < or >= or <= were correct. You received 2 points if you had all 4 boundary tests; 1 point if you had some of the boundary tests; or 0 points if you had no boundary tests.

```
(use-coin-to-pay 25)    ;; should return 'q
(use-coin-to-pay 10)    ;; should return 'd
(use-coin-to-pay 5)     ;; should return 'n
(use-coin-to-pay 1)     ;; should return 'p
```

Finally, we were looking for tests with cases between the coin values. You received a point if you had all of these, or ½ if you only had some.

```
(use-coin-to-pay 22)    ;; should return 'q
(use-coin-to-pay 8)     ;; should return 'd
(use-coin-to-pay 3)     ;; should return 'n
```

You needed to include the expected return value for your tests, or you didn't receive points. If you put down incorrect return values, we took off one point.