**cs1120: Exam 1 Comments**

| Problem | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Average Score** | **9.9** | **7.6** | **7.2** | **9.5** | **9.1** | **8.6** | **6.4** | **7.9** | **8.2** | **8.5** | **5.6** | **5.0** |

Total Grade Distribution:

| Range | Number |
|---|---|
| < 70 | 9 |
| 70-79 | 5 |
| 80-89 | 13 |
| 90-99 | 23 |
| 100-109 | 14 |
| 110+ | 12 |

## Language

**1.** A name in Scheme is any sequence of letters, digits, and special characters (like ! and ?) that starts with a letter or special character. Assume the nonterminals *Letter*, *Digit*, and *SpecialCharacter* are defined to produce the set of all letters, digits, and special characters respectively. Define a BNF grammar that describes the set of valid names in Scheme. Your language should include *a*, *u2*, *!yuck*, and *yikes?47*, but should not include *2a* or the empty string.

*Name ::= Letter MoreChars | SpecialCharacter MoreChars*
*MoreChars* ::= *Letter MoreChars | Digit MoreChars |*
*SpecialCharacter MoreChars* | ε

## Evaluation Rules

**2.** For each of the Scheme expressions below, give the value the expression evaluates to or explain why it is an error. No explanations are necessary, but if the expression evaluates to a procedure you should explain clearly what the procedure is. (Please remember that you are not allowed to use a Scheme interpreter for this exam.)

**a.** (+ 1 1)

```
2
```

**b.** *car*

```
the built-in procedure that extracts the first part of a pair
```

**c.** (*cdr* (*list* 1))

```
null
```

**d.** (**lambda** (*x*) 17)

```
a procedure that takes one input and always outputs 17
```

**e.** ((**lambda** (*p*) (*list-accumulate* (**lambda** (*a b*) (**if** (> *a b*) *a b*)) **0** *p*)) (*list* **1 2 3**))
Assume list-accumulate is defined as in Section 5.4.2 of the book.

```
3
```

## List Procedures

**3.** Define a procedure *list-increment* that takes as input a List of numbers and produces as output a List containing each element in the input List incremented by one. For example,
(*list-increment* (list 1 2))
evaluates to the List (2 3). (This is Exercise 5.17 in the book. You may use any of the procedures defined in Chapter 5 in your answer.)

```
(define (list-increment p)
   (list-map (lambda (x) (+ x 1)) p))
```
Without using list-map:
```
(define (list-increment p)
   (if (null? p) null
       (cons (+ 1 (car p)) (list-increment (cdr p)))))
```

**4.** Define a procedure *list-combiner* that takes as input two Lists of the same length and produces as output a List whose elements are pairs of the corresponding elements in the two input lists. For example,
(*list-combiner* (list 1 2 3) (list 4 5 6))
evaluates to the List containing three cons pairs: ((1. 4) (2 . 5) (3 . 6)). (It is okay if your procedure produces an error if the two input lists have different lengths.)

```
(define (list-combiner p q)
   (if (null? p) null
       (cons (cons (car p) (car q))
             (list-combiner (cdr p) (cdr q)))))
```

Or, the simplest way is to use the built-in map which can operate on multiple lists:

```
(define (list-combiner p q)(map cons p q))
```

## Analyzing Procedures

**5.** What is the asymptotic running time for the list-count-matches procedure defined below:

```
(define (list-count-matches p v)
  (if (null? p) 0
      (if (= (car p) v)
          (+ 1 (list-count-matches (cdr p) v))
          (list-count-matches (cdr p) v))))
```

For full credit, your answer must provide a tight bound on the running time and include a clear and convincing explanation. You may assume the input *v* is a number between 0 and 9999.

The running time is in $\Theta(N)$ where $N$ is the number of elements in $p$. The body of the procedure involves only applications of constant time procedures: cdr, car, null?, + (where the first input is the constant 1), and = (which is constant time because of the assumption that v is bounded). The number of recursive calls is the number of elements in $p$ since each recursive call (on both branches) uses (cdr p) as the new value of p. Thus, there are p applications of a constant time body, so the running time is in $\Theta(N)$ where $N$ is the number of elements in $p$.

**6.** What is the asymptotic running time for the *list-find-first-duplicate* procedure defined below:

```
(define (list-find-first-duplicate p)
  (if (null? p)
      (error "No duplicate found")
      (if (> (list-count-matches (cdr p) (car p)) 0)
          (car p)
          (list-find-first-duplicate (cdr p)))))
```

For full credit, your answer must provide a tight bound on the running time and include a clear and convincing explanation. You may assume all the elements in *p* are numbers between 0 and 9999 but for full credit must explain why an assumption like this is necessary.

The running time is in $\Theta(N^2)$ where $N$ is the number of elements in $p$. From question 5, the running time for list-count-matches is linear in the length of its first input. For each application of list-find-first-duplicate, we evaluate (list-count-matches (cdr p)), so this involves $\Theta(M)$ work where $M$ is the number of elements in p – 1. In the worst case, the first duplicate is at the end of the list (or there is not duplicate), so we need to make $N$-1 recursive calls. Each call involves applications of constant time procedures (we need the assumption that the values of the elements of p are bounded in order for > to be constant time) and the list-count-matches. The average length of the first input to list-count-matches is N/2, so this is $\Theta(N/2)$ work = $\Theta(N)$ for each recursive call. There are up to $N$-1 recursive calls, so the total running time is in $\Theta(N^2)$.

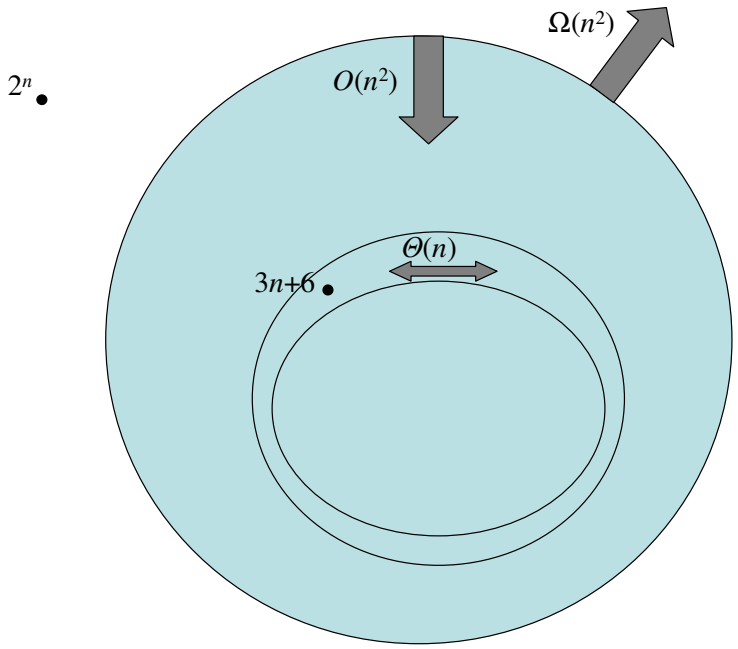
**7.** Draw a picture illustrating the asymptotic growth rates of the following functions and sets of functions:

**a.** $2^n$
**b.** $O(n^2)$
**c.** $\Theta(n)$
**d.** $\Omega(n^2)$
**e.** $3n + 6$

The center of your picture should be the slowest growing functions, and as you move further from the center, functions grow faster (similar to Figure 7.2 in the book). If you are depicting a set, use arrows or color to make it clear what space is included in the set. (There is no need for a fancy drawing. It is fine to hand draw something clear.)

**b, c, and d should be sets; a and e are points.**

**a is a point outside the outermost circle.**
**b is the points inside and on the edge of the $n^2$ circle**
**d is the points outside and on the edge of the $n^2$ circle**
**c is a ring inside the $n^2$ circle**
**e is a point in the ring c.**



## Finding Repeats

One of the main things Colossus did to break the Lorenz cipher was to look for key settings where there were many repeated letters. This was useful since most languages (including German) use repeated letters (for example the two t's in letters) more often than would occur if letters were distributed randomly.

**8.** Define a procedure, *count-repeats*, that takes as input a List of numbers. It produces as output a number that indicates the number of repeated numbers in the input list. We consider a number a repeat if it matches the previous number in the list. So,

(*count-repeats* (list 1 1 2 0)) should evaluate to 1 (the second 1 is a repeat)
(*count-repeats* (list 2 2 2)) should evaluate to 2 (the second and third 2 are repeats)
(*count-repeats* (list 1 2 1 2 1)) should evaluate to 0.

For full credit, your procedure must work correctly for all possible inputs that are Lists of numbers.

```
(define (count-repeats p)
  (if (null? p) 0
     (if (null? (cdr p)) 0
        (if (= (car p) (car (cdr p)))
            (+ 1 (count-repeats (cdr p)))
            (count-repeats (cdr p))))))
```

**9.** Analyze the asymptotic running time of your *count-repeats* procedure. You may assume all numbers in the input list are less than 9999.

My procedure has running time in $\Theta(N)$ where $N$ is the number of elements in $p$. Each recursive call involves only constant-time procedures (because of the assumption about the input list numbers, = can be considered constant time); each call removes on element from the input list $p$.

[These last two questions are meant to be challenging. You are encouraged to answer them, but if you answer the rest of the questions well it is not necessary to answer these questions to achieve "A"-level performance on the exam.]

**10.** Define a procedure, *count-unique*, that takes as input a list of numbers. It produces as output a number that indicates the number of unique numbers in the input list. So,

(*count-unique* (*list* 1 1 2 0)) should evaluate to 3.
(*count-unique* (*list* 2 2 2)) should evaluate to 1.
(*count-unique* (*list* 1 2 1 2 1)) should evaluate to 2.

For full credit, your procedure must work correctly for all possible inputs that are Lists of numbers.

There are many different ways to define count-unique, some of which use procedures from earlier exam questions.

The easiest definition is to observe that if the list is sorted, then the non-unique numbers will be adjacent. So, we can use count-repeats (from question 8) to count the repetitions, and the number of unique elements is the length of the list minus this:

```
(define (count-unique p) (- (length p) (count-repeats (sort p <))))
```

Perhaps a more straightforward, but much longer solution is to remove the duplicates from the list:

```
(define (extract-matching el p)
  (if (null? p) null
      (if (= el (car p)) (extract-matching el (cdr p)) p)))

(define (remove-duplicates p)
  (if (null? p) null
      (cons (car p) (remove-duplicates (extract-matching (car p) (cdr p))))))

(define (count-unique p) (length (remove-duplicates (sort p <))))
```

If the list isn't sorted, it is necessary to search through the whole list for the duplicates to remove. Since the list is sorted, our extract-matching procedure stops after finding the first matching element. If the list is not sorted, we would instead need to do,

```
(define (extract-matching el p)
  (if (null? p) null
      (if (= el (car p))
          (extract-matching el (cdr p))
          (cons el (extract-matching el (cdr p))))))
```

**11.** Analyze the asymptotic running time for your *count-unique* procedure. Carefully define any variables you use and explain any assumptions needed for your analysis to be correct.

Θ(*N* log *N*) where *N* is the number of elements in *p*. We need to assume all the elements of *p* are within a bounded range, so the < and = procedures have constant running times. (Note that this assumption actually enables an asymptotically faster solution as discussed in class.)

For the simplest definition, the body is: `(- (length p) (count-repeats (sort p <))`
Note that there are no recursive calls, so we just need to sum the running-time of all the procedure applications: length is in Θ(*N*), count-repeats is in Θ(*N*) (from question 9), sort is in Θ(*N* log *N*), and – and < are constant time given the assumption about the values of the elements of *p* being in a bounded range. The sum of Θ(*N*) + Θ(*N*) + Θ(*N* log *N*) is in Θ(*N* log *N*). Eventually, only the fastest growing term matters.

Analyzing the second definition is more complex. The worst case running time for extract-matching is in Θ(*M*) where M is the number of elements in the second input. The remove-duplicates procedure involves up to *W* recursive calls where *W* is the length of its input, and each call involves a call to extract-matching. This looks like the running time for extract-matching is in $\Theta(W^2)$ but its not. The reason for this is the size of the input to the recursive call of remove-duplicates is reduces by the number of elements removed by extract-matching. The running time for extract-matching scales linearly with the number of elements removed. Hence, the overall running time for remove-duplicates is in Θ(*W*) where *W* is the length of its input.

count-unique applies three procedures: sort, remove-duplicates, and length. The input to sort is the list p, so the running time for sort is in Θ(*N* log *N*) where *N* is the number of elements in *p*. It produces a list of length N, which is the input to remove-duplicates, so the running time of the remove-duplicates application is in Θ(*N*). The input to length is no longer than *N* elements, so the running time for the length application is in *O*(*N*). Thus, the total running time for count-unique is in Θ(*N* log *N*) + Θ(*N*) + O(*N*). The fastest growing term is the first one, so the total running time is in Θ(*N* log *N*).

If your count-unique procedure didn't sort the elements first, its running time is probably in $\Theta(N^2)$. This is the case for our definition using the second version of extract-matching, since this has running time in Θ(*N*) and would be evaluated up to *N* times.