CSC324 - Principles of Programming Languages
Summer 2003

Department of Computer Science
University of Toronto

Midterm

Duration: 75 minutes
External aids allowed: None

Make sure that your examination booklet has 8 pages of questions (excluding this one). Write your answers in the space provided. Write legibly. Good luck.

Surname:_____ **First Name:**_____

Student #:_____

1._____**/20**

2._____**/20**

3_____**/20**

4_____**/20**

5_____**/20**

Total_____**/100**

1.

(1.a) [5 points]
Java is
(a) an island in Indonesia
(b) a slang term for coffee
✓ **a popular programming language**
(c) a small town in South Dakota

(1.b) [5 points]
Assume function (member? x L) is available and returns #t *if and only if* x is a member of list L. Here is an attempted definition for a new function rp?

```
(define (rp? L)
        (cond  ((null? L) #f )
               ((member? (car L) (cdr L)) #t)
               (else (rp? (cdr L) ) )
```

What is it intended to do?

Answer[-3 ➔ described the code without explicitly stating that it is checking for duplicates, -5 ➔ rewrote the code using English statements]:
Checks whether there are duplicates or not

(1.c) [5 points]
Briefly describe the **Three Deadly Pointer Sins** (common programming errors caused by pointers).

Answer: [pick any three, -1.5 ➔ for missing (each)]
Dangling Pointer/reference:
Storage pointed to is free, but pointer is not set to null. Then, you access storage whose value is not meaningful
Garbage:
Pointer itself is freed, but heap locations pointed to are not freed.
Memory Leaks:
Gradual loss of available computer memory when a program repeatedly fails to return memory that it has obtained for temporary use.
Un-initialized Pointer:
Attempting to de-reference a pointer that has not been initialized or contains the NULL pointer.

(1.d) [5 points]
List four design issues relating to counter-controlled loops.
Answer: [pick any four, -1.25 ➔ for missing (each)]
What is the type of the loop variable? What is the scope of the loop variable? What value does the loop variable contain on loop termination? Is it possible to alter the loop variable in the body of the loop? Is there a special closing statement? Does the loop body have compound-statements/single statement/block? Should the test for the loop completion be at the top or bottom of the loop? Should the loop parameters be evaluated only once, or for every iteration?

2.

(2.a) [10 points]
Consider the following grammar where uppercase letters indicate non-terminals and lowercase letters indicate terminals:

S → a S c B | A | b
A → c A | c
B → d | A

Which of the following sentences are in the language defined by this grammar? If the sentence is in the language, show a parse tree.

acccbd

Answer: [2.5 points]
Clearly S can begin with a and end with b. But how can we get "bd"? The only way to get a b is through the first S production but that requires c to appear immediately before d so the first string is not generated by this grammar.

aabcdcd

Answer:[2.5 points]
This rule begins with a and ends with "cd" so it might be generated but it has "abcd" in the middle. But wait, "abcd" is a string generated by the grammar. That means S=>abcd. Here is a rightmost derivation:

```
S    => aScB
     => aaScBcB      // S=>aScB
     => aabcBcB      // S=>b
     => aabcdcB      // B=>d
     => aabcdcd      // B=>d
```

Here is the parse tree drawn with all the "children" of a node indented on following lines:

```
S
  a
  S
    a
    S
      b
    c
    B
      d
  c
  B
    d
```

acd

Answer:[2.5 points]
We can easily get from S to aScd but there is no "epsilon production" for S. We can't make the S simply go away without being replaced by some terminal so the string acd is not generated by this grammar.

accc

Answer: [2.5 points]
The string is easily generated by this leftmost derivation:

```
S   => aScB    // S=>A
    => aAcB    // A=>c
    => accB    // B=>A
    => accA    // A=>c
    => accc
```

Here is the parse tree:

```
S
    a
    S
        A
            c
    c
    B
        A
            c
```

(2.b) [10 points]
Using the following grammar, with non-terminals depicted with capital letters, answer the questions with True or False.

G ::= S
S ::= aS | aX
X ::= bc | bXc

Answers: [each 2.0 points, count any correct 5]

FALSE abcbc is a sentence in this language.

TRUE abbbccc is a sentence in this language.

TRUE This is a context-free grammar.

TRUE aX is a sentential form of this language.

FALSE This grammar is regular.

TRUE G → S → aX → abXc is the partial parse of a string in the language.

3.

(3.a) [5 points]
Consider overloading the + operator in a language like C to allow the programmer to add pointers. Is this orthogonal or non-orthogonal? Why?

Answer: [2 points for whether it is orthogonal or not, 3 points for explanation]
From the text: "A relatively small number of primitive constructs can be combined in a relatively small number of ways to build the control and data structures of the language. Furthermore, every possible combination of primitives is legal and meaningful."

Lack of orthogonality gives rise to exceptions or special cases that are hard to remember. Too much or thogonality gives rise to weird combinations that are rarely used. Simple programming errors may end up invoking some feature the programmer is unfamiliar with. So too little or too much orthogonality reduces reliability.

Overloading '+' in C is NON-ORTHOGONAL because adding pointers actually means something more complicated than just adding numbers.

(3.b) [15 points]
Give the access formula for a one-dimensional array with lower and upper indices LB, and UB, base BASE, element size ESIZE. Assume you are accessing element i. In other words, show how to compute the address of element a[i] for an array declared like this:

a: array [LB .. UB] of element;

where a is stored in memory at address BASE and sizeof(element) is ESIZE.
Show how the expression can be rewritten to save runtime computation.

Answer: [10 points for formula, 5 points for optimization]
UB is not needed.
address of a[i] = BASE + (i-LB) * ESIZE;
This involves three operations: +, -, and *

Rewriting we get:

address of a[i] = (BASE - LB*ESIZE) + i*ESIZE;

The first part can be computed once. Subsequent array index computations require only two operations: - and *.

4.

(4.a) [5 points]
The *norm* of a vector is the square root of the sum of the squares of its elements. Write a Scheme function named **(norm …)** that takes any vector of real numbers as its argument and returns the norm of that vector**.** Here are some examples:

```
> (norm '#(9 12 8))
17
> (norm '#(1 2 3 4))
5.477225575051661
> (norm '#())
0
```

Answer:[-1 ➔ no error checking, -0.5 ➔ incomplete error checking, -1 ➔ syntax error (each), -1 ➔ efficiency/style, 1➔ some idea for solution]

```
;; The NORM procedure computes the square root of the sum of the squares
;; of the elements of a given vector.

(define norm
  (lambda (vec)

    ;; Make sure that VEC is a vector of numbers.

    (if (not ((vector-of number?) vec))
        (error 'norm "The argument must be a vector of numbers"))
    ;; Traverse the vector (from right to left, in this case), squaring
    ;; each element and adding the result to a running total.  After all
    ;; of the elements have been processed, extract and return the square
    ;; root of the running total.

    (let loop ((remaining (vector-length vec))
               (total 0))
      (if (zero? remaining)
          (sqrt total)
          (let ((next (- remaining 1)))
            (loop next (+ total (square (vector-ref vec next)))))))))

;; The VECTOR-OF procedure takes a predicate PRED as argument and returns a
;; procedure that takes any object as argument and determines whether it is
;; a vector in which PRED correctly characterizes every element.  (If it is
;; a vector that has no elements, the procedure returned by VECTOR-OF
;; returns #T, on the theory that PRED is ``vacuously true" of the
;; vector's elements.)

(define vector-of
  (lambda (pred)
    (lambda (obj)
      (and (vector? obj)
           (let loop ((remaining (vector-length obj)))
             (or (zero? remaining)
                 (let ((next (- remaining 1)))
                   (and (pred (vector-ref obj next))
                        (loop next)))))))))

;; The SQUARE procedure takes any number and returns the result of
;; multiplying it by itself.

(define square
  (lambda (n)
    ;; Make sure that N is a number.
    (if (not (number? n))
        (error 'square "The argument must be a number"))
    ;; Square it.
    (* n n)))
```

(4.b) [15 points]
Write a recursive Scheme function **(maximize …)** that takes two lists of real numbers as arguments, compares the numbers that are in corresponding positions on the two lists, and returns a single list containing at each position, the larger of the numbers compared for that position. For example:

```
> (maximize '(1 2 3 4 5) '(5 4 3 2 1))
(5 4 3 4 5)
> (maximize '(-3 -8 12 0 6 -4) '(-7 3 9 6 6 -2))
(-3 3 12 66 -2)
> (maximize '(5 5 5) '(4 4 4))
(5 5 5)
```

If the given lists are of different lengths, the result should always be as long as the longer of them, and the longer list should provide the result value for any position after the end of the shorter list. For example:

```
> (maximize '(1 2 3 4 5) '(8 7 6))
(8 7 6 4 5)
> (maximize '(-3 4) '(3 2 -7))
(3 4 -7)
> (maximize '(-1 -1 -1 -1) '())
(-1 -1 -1 -1)
```

Answer:[-10 ➔ not recursive, -3 ➔ no error checking, -1/-2 ➔ poor error checking, -5 ➔ not handling lists of different lengths properly, -1 ➔ syntax error (each), -2 ➔ efficiency/style, -5 ➔ improper lists, -5 ➔ missing base case, 1-4 ➔ some idea for solution]

```
;; The MAXIMIZE procedure takes two lists of real numbers as arguments,
;; compares the numbers that are  in corresponding positions on the two
;; lists, and returns a single list containing, at each position, the
;; larger of the numbers compared for that position.  If the given lists
;; are of different lengths, the result is always be as long as the longer
;; of them, and the longer list provides the result value for any position
;; after the end of the shorter list.

;; In this implementation, MAXIMIZE is a husk; it confirms that its
;; arguments are both lists of real numbers, then invokes MAXIMIZE-KERNEL
;; to carry out the actual construction of the result list.

(define maximize
  (lambda (ls-1 ls-2)
    (if (or (not (list-of-reals? ls-1))
            (not (list-of-reals? ls-2)))
        (error 'maximize "both arguments must be lists of real numbers"))
    (maximize-kernel ls-1 ls-2)))

;; If either of the arguments to MAXIMIZE-KERNEL is an empty list, the
;; other argument is returned unchanged.  Otherwise, MAXIMIZE-KERNEL
;; compares the first element of one list to the first element of the
;; other.  Whichever is larger is prepended to the result of the recursive
;; call that compares elements in subsequent corresponding positions of the
;; lists.

(define maximize-kernel
  (lambda (ls-1 ls-2)
    (cond ((null? ls-1) ls-2)
          ((null? ls-2) ls-1)
          ((< (car ls-1) (car ls-2))
           (cons (car ls-2) (maximize-kernel (cdr ls-1) (cdr ls-2))))
          (else
           (cons (car ls-1) (maximize-kernel (cdr ls-1) (cdr ls-2)))))))

;; The LIST-OF-REALS? procedure takes any Scheme value and determines
;; whether it is a list of real numbers.  The empty list qualifies as a
;; list of real numbers; so does any pair in which the first element is
;; a real number and the rest of the list is itself a list of real numbers.

(define list-of-reals?
  (lambda (arg)
    (or (null? arg)
        (and (pair? arg)
             (real? (car arg))
             (list-of-reals? (cdr arg))))))
```

5.

(5.a) [5 points]
Consider the following function

```
(define meta-mystery
        (lambda (expression)
                (list 'begin
                (list 'write (list 'quote expression))
                (list 'display " ==> ")
                (list 'write expression)
                (list 'newline))))
```

What is the value of the function call (meta-mystery '(+ 3 5))?

Answer: [5 points for writing it like this, -2 if written on multiple lines]
(begin (write (quote (+ 3 5))) (display " ==> ") (write (+ 3 5)) (newline))

(5.b) [5 points]
What does the following Scheme expression compute? Why?

```
(let ( (+    (lambda (x y) (+ y 2) ) ) )
        (let ( (+   (lambda (x y) (+ y x) ) ) )
                (+ 5 (+ 2 1) )
        )
)
```

Answer:[3 points for answer, 2 points for explanation]
The output is 7
The + operator is defined twice in the outer and inner let. When evaluating the inner let, the definition from the first one is used.

(5.c) [5 points]
List the six attributes associated with programming language *variables*.

Answer: [count any correct 5, -1 for missing]

Name
Address
Value
Type
Lifetime
Scope

(5.d) [5 points]
List three problematic issues with imperative programming languages.

Answer: [pick any correct 3, -1.5 for missing]

Von Neumann bottleneck, minimal support for parallel programming
Assignment
Side-effects
State-based transformations
No referential transparency