Name (1 point): ______________________________

ID number (1 point): ______________________________

CS 212, Spring 1997

Exam 2

April 17, 1997

Problem	0	1	2	3	4	5	Total
Grader	$\overset{\infty}{\aleph}\,\frac{\frac{\top\,\Im}{\nabla\,\Im}}{\Im}{}^{\mho}_{\flat}$						
Grade	2						

There are five (5) problems on this exam. Please check now that you have a complete exam booklet with ten (10) numbered pages. Please write your name or id number on each page of the exam. Be sure to try all of the problems, as some are more difficult than others (i.e., don't waste a lot of time on a problem that is giving you a hard time – move on to another problem and then return to it later).

There is space provided to answer each question. You may request extra paper if necessary. *Do not put any answers on the backs of pages* (THEY WILL NOT BE GRADED), get an extra piece of paper if you need more space.

To help ensure that you don't accidentally miss any of the questions, we have marked those sections where an answer is requested with a $\Longleftarrow$ in the right margin.

The staff reserves the right to ignore illegible answers. "Pretty printing" (proper indentation) of your code will aid in the grading process.

Note that this exam covers material through lecture 18. Solutions using macros will $\boxed{\text{not}}$ be accepted.

This test is closed book and closed note! The Dylan cheat sheet will be available.

1. (20 points) **Object-oriented programming**

   Consider the following definitions.

```
(define-class <thing> (<object>)
 (size type: <number> init-keyword: size: init-value: 100)
 (location type: <object> init-keyword: location:
   init-value: 'nowhere))

(define-class <animal> (<thing>)
 (name type: object init-keyword: name: init-value: 'anonymous)
 (hunger type: <number> init-keyword: hunger: init-value: -1))

(define-class <vegetable> (<thing>)
 (nutritional-value  type: <number> init-value: 0))

(define-class <mineral> (<thing>)
 (worth type: <number> init-value: 0))

(define-class <venus-flytrap> (<animal> <vegetable>)
 (favorite-meals type: <list> init-value: '(flies)))

(define-class <fly> (<animal>))

(define popeye (make <animal> size: 150 name: 'popeye))

(define spinach (make <vegetable> location: 'grocery-store))

(define milo (make <venus-flytrap> name: 'milo))
```

   (a) List the slots of `popeye`, `spinach` and `milo`, together with their values.

popeye	spinach	milo

(b) Write a generic function `edible?`, which takes two arguments, an `eater` and an `eaten`. The generic function will return `#t` if the eater can eat the eaten. In general, an arbitrary `<thing>` cannot be eaten. An `<animal>` can eat a `<vegetable>` (but not the reverse). However, a `<venus-flytrap>` can eat a `<fly>`.

2. (30 points) **Data Structures**

One sometimes inconvenient feature of lists in Dylan is that it is only possible to iterate through the list in one direction. We can get around this by implementing two-way lists. Consider that regular lists are built from cells (ie. cons cells) containing two pointers, where the first (car) points to a value of the list and the second (cdr) points to the next cell. We can extend this idea to handle two-way lists by adding an extra pointer to our list element. Consider the following code:

```
(define-class <triple>
  (previous type: <object> init-keyword: prev:)
  (value type: <object> init-keyword: val:)
  (next type: <object> init-keyword: next:))
(define (make-triple <method>)
  (method ((p <object>) (v <object>) (n <object>))
    (make <triple> prev: p val: v next: n)))
```

Using this code, we can now implement a mutable double ended queue (deque) which supports the following operations such that they take $O(1)$ (constant) time.

`(new-mutable-deque)` : Create a new empty mutable deque.

`(insert-front! val deq)` : Put val at the front of the deque.

`(insert-end! val deq)` : Put val at the end of the deque.

`(extract-front! deq)` : Remove value at the front of the deque and return it.

`(extract-end! deq)` : Remove value at the end of the deque and return it.

The implementation should behave as follows:

```
(define (a <deque>) (new-mutable-deque))
(define (b <deque>) (new-mutable-deque))
(insert-front! 5 a)
(insert-front! 4 a)
(insert-end! 'fred b)
(insert-end! 7 a)
(insert-front! 'barney b)
(extract-front! a) ⟹ 4
(insert-end! 'beavis b)
(extract-end! a) ⟹ 7
(insert-front! 2 a)
(extract-end! a) ⟹ 5
(extract-end! a) ⟹ 2
(extract-front! b) ⟹ barney
(extract-front! b) ⟹ fred
(extract-front! b) ⟹ beavis
```

(a) Before writing any code, draw an illustration of your data structure for a **nonempty** `<deque>`, using `<triple>`s. Your data structure must support all constant time operations.

(b) Write code for the new class `<deque>`, and the three (3) procedures **new-mutable-deque**, **insert-front!**, and **extract-end!**. You *do not* need to write code for the other procedures, since they are similar, but you must ensure that your design would support these operations correctly and in constant time.

```
(define-class <deque>
```

```
(define new-mutable-queue
   (method ((o <object>) (q <deque>))
```

```
(define insert-front!
    (method ((o <object>) (q <deque>))
```

```
(define extract-end!
    (method ((q <deque>))
```

3. (14 Points) In this problem you are to write a procedure `memoize` which takes as input a procedure `f` of one argument and returns a procedure that produces the same value as `f` on any input but which only computes `f` once for any particular value. That is, the first call to the resulting procedure with a given argument computes the value of `f` on that argument and then stores and returns the value. Subsequent calls with that same argument simply look up and return the stored value.

   For instance, consider the following procedure defined using `memoize`,

```
(define test
  (memoize (method ((x <number>))
             (print 'computing)
             (* x x))))
```

   The following calls then result in the output shown in the following transcript:

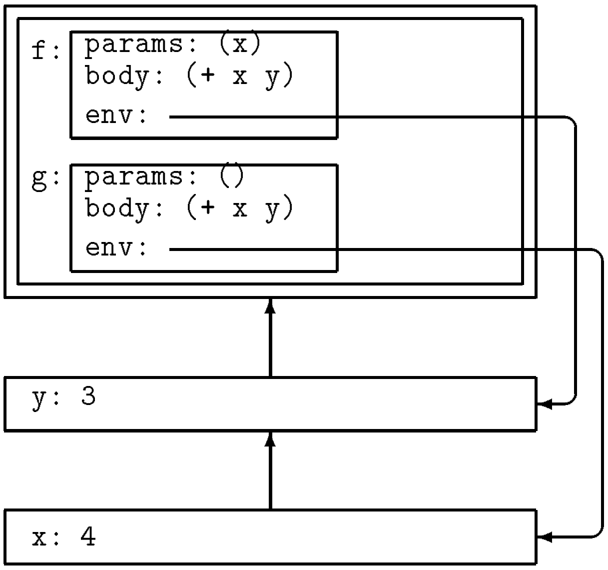
```
? (test 5)
computing
==> 25

? (test 4)
computing
==> 16

? (test 5)
==> 25
```

   Write the procedure `memoize`.

4. (15 points) Suppose we have just executed the following code:

```
(define (f <function>) (method () 0))
(define (g <function>) (method () 0))
```

Give an expression that when evaluated will result in the environment illustrated below.



The global environment is the frame at the top with the double border.

HINT: Your expression should begin with `(bind...`, like this:

```
(bind
```

Question 5: (15 points)

Math wiz Prudence Juris wins the Fields Medal by proving the following theorem:

Theorem 1 *All students in CS 212 will get the same grade.*

Proof: We prove the theorem by induction.

Base case: Consider a set of 1 CS 212 student. Certainly the student gets the same grade as herself.

Induction hypothesis: Let $n \geq 1$. Suppose that for any set A of n CS212 students, all students in set A will receive the same grade.

Induction step: Suppose the induction hypothesis holds for some $n \geq 1$. We need only show that for any set B of $n+1$ CS212 students, that all students in set B will receive the same grade.

(Part 1) Take the set B. It has $n+1$ students. Take one, say s, out, to obtain a set $A = B - \{s\}$. Now, set A contains n students so by the induction hypothesis, all students in A will recieve the same grade. Call this grade g_s. Now, put s back in to obtain B again.

(Part 2) Pick another student t of B, different from s. Remove t from B to obtain another set C. Now, C has n elements, so that all students in C will receive the same grade, by the induction hypothesis. Call this grade g_t.

We can show that $g_s = g_t$. Pick any student r in B different from s and t. Then, r will receive grade g_s in part 1 and grade g_t in part 2. Since r receives identically one grade, $g_s = g_t$. Hence students t, s and r all receive the same grade. Since the choices of students were arbitrary, we conclude that all students in CS212 will receive the same grade. □

Prudence is very proud of her result, and hopes to get an $A+$ in CS212. But she is dismayed when her classmate, lazy, slothful Nick Nurd proves the following lemma

Lemma 1 *I (Nick) will flunk CS212.*

Proof: By construction. I won't study, I'll get roaring drunk before the test, dress worse than the CS212 staff, ride my motorcycle into the test halfway through, and enrage the TAs by mixing up their names on purpose. □

This results in the following unfortunate

Corrollary 1 *All students will fail CS212.*

Proof: By Nick's Lemma 1, one student (Nick) will get an F. Hence, by the theorem 1 of Juris (above), all students will get an F. □

Prudence is enraged and saddened. All that work down the drain because of one shiftless, no-account bum! She goes to consult her favorite logic professor, Letcher S. Hazbin. Hazbin consoles her: "Well, you *could* try to find a flaw in Nick's lemma... But that would involve reforming him so that he'll pass. An odious task! If I were you, I'd look for a bug in *your* argument. If you find one, then he can flunk, and you can still get an A!"

(a) Is there a flaw in her proof? Explain briefly in good English. If there is a flaw, identify it and explain. You may assume that Nick's lemma is true.

(b) Assume Nick's lemma is true. Will all students fail CS212? Explain.