CS 3 (Clancy) Final exam
May 17, 1993

THIS COPY HAS ANSWERS DISPLAYED.
Read and fill in this page now.
Do NOT turn the page until you are told to do so.
Your name:
(please print last name, first name) Name of the person sitting to your left:
Name of the person sitting to your right:
Total: /60
Problem 0
Problem 1
Problem 2
Problem 3
Problem 4
Problem 5
Problem 6

This is an open-book test. You have approximately three hours to complete it; time estimates accompanying each problem suggest a pace to finish in three hours. You may consult any books, notes, or inanimate objects other than Lisp interpreters available to you. To avoid confusion, 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.

This exam comprises 30% of the points on which your final grade will be based. Partial credit will be given for wrong answers. You are not to use setf within functions. Any other Lisp construct described in Touretzky chapters 1-8, any of the case studies, or any of the CS3 handouts is ok, though. In particular, either recursion or applicative operators is allowed in a solution unless otherwise specified. Anywhere you are directed to write a function, your solution may include auxiliary functions. You need not rewrite a function that appears in Touretzky, in any of the case studies, or in any of the CS 3 handouts; merely cite the page in Touretzky, the case study and page number or appendix, or the handout in which the function appears.

Your exam should contain 6 problems (numbered 0 through 6) on 13 pages. Please write your answers in the spaces provided in the test; in par- ticular, we will not grade anything on the back of an exam page unless we are clearly told on the front of the page to look there. Relax! this exam is not worth having heart failure about.

Problem 0 (3 points, 5 minutes)
You earn the points for this problem if you have done the following:

- you have put your name on each page of this exam, and provided the information requested on the first page;

- you have completed and handed in the *CS 3 Final Survey* (handed out in lecture; copies of the latter are available from Mike Clancy or the t.a.s);
- you have completed an interview.

Check with Mike Clancy or the t.a.s to make sure your survey submission and your interview have been recorded.

Background for problems 1 and 2
Problems 1 and 2 refer to a state results list. The elements of a legal state results list are themselves three-element lists of the following form:

- the first element is an atom, the name of the state (multi-word names are combined with hyphens, as in new-york or south-dakota);
- the second element is the number of electoral votes from the state (how many senators and congresspeople it has);
- the third element is an atom, the last name of the candidate that won the electoral votes from the state.

The empty list is a legal state results list. A legal state results list also has at most one entry for each state. An example is the following.

```
((california 54 clinton)
 (new-york 33 clinton)
 (texas 32 bush)
 (illinois 22 clinton)
 (florida 25 bush)
 (ohio 21 clinton)
 (pennsylvania 23 clinton) )
```

Problem 1 (5 points, 15 minutes)
Consider the following code. (Assume that remove-duplicates returns the result of removing duplicate elements from its input.)

```
(defun mystery1 (state-results-list)
        (mystery2 (mapcar #'third state-results-list)) )
(defun mystery2 (name-list)
        (defun mystery3 (name1)
                (length
                        (remove-if-not
                                #'(lambda (name2) (equal name1 name2))
                                name-list) ) )
        (mapcar
                #'(lambda (name) (list name (mystery3 name)))
                (remove-duplicates name-list) ) )
```

Part a (2 points)
Describe in English what mystery3 returns when given as input a candidate name along with the list of names provided to mystery2.

the number of times the given name appears in the name list

1 pt for count
1 pt for what kind of count

Part b (3 points)
Describe in English what mystery1 returns when given as input a state results list.

a table, for each candidate who won some state, pairing the candidateÕs name with how many states he/she won

1 point for some hint of list of lists
1 point for name as table key
1 point for # states won as table value

Problem 2 (10 points, 30 minutes)
Consider the following code intended to check that its input is a legal state results list.

```
(defun legal? (x)
        (and
                (listp x)
                (correct-state-names? (mapcar #'first x))
                (correct-electoral-votes? (mapcar #'1st+2nd x))
                (correct-candidate-names? (mapcar #'third x))
                (not (contains-duplicates? (mapcar #'first x))) ) )
(defun 1st+2nd (L)
        (list (first L) (second L)) )
```

Comments for functions whose code is not supplied are as follows:
- Correct-state-names? assumes that its input is a list. It returns true when the input list is empty or when each element of the input is the name of a state, and returns nil otherwise.
- Correct-electoral-votes? assumes that its input is a list of two-element lists. It returns true when the input list is empty or when each element of the input is a state paired with the correct number of electoral votes for that state, and returns nil otherwise.
- Correct-candidate-names? assumes that its input is a list. It returns true when the input list is empty or when each element of the input is the name of a candidate, and returns nil otherwise.
- Contains-duplicates? works as described in the Mastermind case study.

Part a (2 points)

Give an example of an illegal state results list for which legal? returns true, or explain why there can be no such input.

some element has more than three elements (i.e. junk past the candidate name)
2 pts

Part b (2 points)
Give an example of an illegal state results list for which legal? crashes (i.e. produces an error message), or explain why there can be no such input.

some element is an atom
2 pts

Part c (6 points)
The code for legal? is repeated below. Add whatever code is necessary to fix it so that it returns true for all legal state results lists and returns nil for any input that's not a legal state results list. Indicate clearly how legal? is to be modified, and provide code for whatever auxiliary functions you call.

```
(defun legal? (x)
        (and
                (listp x)
                (correct-state-names? (mapcar #'first x))
                (correct-electoral-votes? (mapcar #'1st+2nd x))
                (correct-candidate-names? (mapcar #'third x))
                (not (contains-duplicates? (mapcar #'first x))) ) )
(defun 1st+2nd (L)
        (list (first L) (second L)) )

add after (listp x): (all-lists? x)
        (defun all-lists? (L)
                (every #'listp L) )
add after all-lists?: (all-elems-length-3? x)
        (defun all-elems-length-3? (L)
                (every #'(lambda (L2) (= (length L2) 3)) L) )
```

3 pts each modification, for a total of 6

Background for problems 3 and 4
Problems 3 and 4 concern functions one might use to write a compress function, the inverse of the uncompress function you wrote for homework assignment 9. Compress will be given a single list of 0's and 1's as input. It will return the result of converting each sequence of two or more adjacent identical elements into the list (n x), where x is the duplicated element and n is the number of x's in the sequence. For example, (compress '(1 1 0 1 0 0 0 0 1)) should return the list ((2 1) 0 1 (4 0) 1).

Recall the functions result-of-compression? and uncompress that you wrote for assignment 9. For all lists L that contain only 0's and 1's, the following should be true:

```
(and
        (result-of-compression? (compress L))
        (equal L (uncompress (compress L))) )
```

Problem 3 (17 points, 45 minutes)
One way to write compress would be to use a recursive function compress-helper as follows:
; Given a list L of 0's and 1's, return the result ; of compressing the list as described above.

```
(defun compress (L)
        (if (null L)
                nil
                (compress-helper (list (first L)) (rest L)) ) )
```

; So-far is a nonempty list containing either all 0Õs ; or all 1Õs; it is the most recent sequence of 0Õs ; or 1Õs seen so far. L is the list of 0Õs and 1Õs ; that have not yet been processed. ; Return the result of compressing the list ; (append so-far L). (defun compress-helper (so-far L) ...) Part a (3 points) Suppose that compress-helper correctly performs according to its comment. Indicate the result of evaluating each of the following expressions. expression result (compress-helper '(0) '( )) (0) (compress-helper '(0 0 0) '( )) ((3 0)) (compress-helper '(0 0) '(0 0 1 0 1 1)) ((4 0) 1 0 (2 1)) (compress-helper '(1 1) '(0 0 1 0 1 1)) ((2 1) (2 0) 1 0 (2 1)) (compress-helper '(1) '(0 0 1 0 1 1)) (1 (2 0) 1 0 (2 1)) -1 each wrong answer up to 3 Part b (14 points) Fill in the blanks in the framework for compress-helper below. Your code may include recursive calls, but not any use of applicative operators. ; So-far is a nonempty list containing either all 0Õs or all 1Õs; ; it is the most recent sequence of 0Õs or 1Õs seen so far. ; L is the list of 0Õs and 1Õs that have not yet been processed. ; Return the result of compressing the list (append so-far L). (defun compress-helper (so-far L) (cond ((null L) ; example: (compress-helper '(0) '( )) ; example: (compress-helper '(0 0 0) '( ))) (if (= (length so-far) 1) so-far (list (list (length so-far) (first so-far))))) 4 pts 1 for the test 1 for (length so-far) = 1 2 for (length so-far) > 1 ((equal (first so-far) (first L)) ; example: (compress-helper '(0 0) '(0 0 1 0 1 1))) (compress-helper (cons (first L) so-far) (rest L))) 2 pts 1 for each input ((> (length so-far) 1) ; example: (compress-helper '(1 1) '(0 0 1 0 1 1))) (cons (list (length so-far) (first so-far)) (compress-helper (list (first L)) (rest L)))) 5 pts (T ; example: (compress-helper '(1) '(0 0 1 0 1 1))))) (append so-far (compress-helper (list (first L)) (rest L)))) 3 pts another breakdown for the recursion cases: (10 pts) recursive call (5 pts) new so-far (first so-far) = (first L)) -- 2 pts not equal -- 2 pts new L -- 1 pt consistent misuse of cons, list, or append here: deduct no more than 2 using the result of the recursive call (5 pts) (equal (first so-far) (first L)) -- 1 pt (> (length so-far) 1) -- 2 pts (= (length so-far) 1) -- 2 pts consistent misuse of cons, list, or append here: deduct no more than 2 (may overlap with base case) Problem 4 (16 points, 50 minutes) Another way to write compress would be to make use of a function called group (similar to the code used in the Statistics case study to determine the mode of a list of numbers). Group would take a list L as input, and return the result of grouping adjacent identical elements in L. Thus (group '(AÊBÊBÊB AÊCÊ0Ê0Ê1Ê1Ê1Ê1ÊX)) would return the list ((A) (BÊBÊB) (A) (C) (0Ê0) (1Ê1Ê1Ê1) (X)). Part a (6 points) Use the group function to write compress. Do not use recursion.

You may use auxiliary functions, but these should not use recursion either. (defun compress (L) (mapcar #'compress-help (group L))) (defun compress-help (group) (if (= (length group) 1) (first group) (list (length group) (first group)))) 6 pts 2 for correct use of mapcar and group (-1 for misquoting) 2 for the case where group length = 1 2 for the case where group length > 1 Two functions that might be used to write group are the following: ¥ Pos-of, given as inputs a predicate pred and a list L, returns the position of the first element of L that satisfies pred. (Pos-of will thus be an applicative operator that resembles find-if.) Positions start at 0. If no elements of L satisfy pred, pos-of returns the length of L. ¥ Pos-of-different, given as input a nonempty list L, returns the position of the first element in L thatÕs not the same as (first L). As with pos-of, positions start at 0. If all elements of L are identical, pos-of-different returns the length of L. Part b (4 points) Complete the definition of pos-of below. Your solution may use either recursion or applicative operators, and may include auxiliary functions. (defun pos-of (pred L) (defun pos-of (pred L) (cond ((null L) 0) ((funcall pred (first L)) 0) (T (+ 1 (pos-of pred (rest L)))))) they might also redo the my-find-if function they wrote for the recursion lab as follows: (defun pos-of (pred L) (- (length L) (length (my-find-if pred L)))) they have to supply the code for my-find-if also. 4 points 2 for getting the funcall right -1 if they used #' -1 if the second input was (list (first L)) or something else slightly varying from reality -2 if they didn't use funcall at all 2 for the rest of the recursion -1 for each wrong base case value -1 for wrong recursive call Part c (3 points) Fill in the blank in the framework for pos-of-different below. Assume that pos-of works as specified regardless of what you wrote for part b. You may define auxiliary functions; indicate clearly whether their definitions should appear inside or outside pos-of-different. (defun pos-of-different (L) (pos-of L)) (defun pos-of-different (L) (pos-of #'(lambda (x) (not (equal x (first L)))) L)) (defun pos-of-different (L) (defun helper (x) (not (equal x (first L)))) (pos-of #'helper L)) 3 pts partial credit: 2 for essentially correct solution that contains misquoting or misparenthesizing (don't deduct points for both) 1 for an incorrect solution that involves some use of not and equal Part d (3 points) The grouping code for computing the mode in the Statistics case study contains a function group-element that might be used with reduce as follows: (reduce #'group-element '((0) 0 1)) What sequence of applications of group-element will result from evaluating this expression? That is, if you typed (evaluate (reduce #'group-element '((0) 0 1))) to the Listener, and then hit return once in the Evaluation Modeler to expand the reduce, what would appear in the Evaluation Modeler window? (Note that you donÕt need the Statistics code to answer this question.) (group-element (group-element '(0) 0) 1) 3 pts partial credit: 1 point for having two calls to group-element, one nested with the other, but in the wrong order or with inputs in the wrong order, e.g. (group-element '(0) (group-element 0 1)) Don't take off quoting or parenthesizing points for this. 1 point for three correctly sequenced calls, one with a nil, e.g. (group-element (group-element (group-element '( ) 0) 0) 1) Again, no quoting or parenthesizing deductions here. 2 points for an essentially correct solution with the list misquoted or misparenthesized. Problem 5 (6 points, 20 minutes) Give a pattern and object that, when provided as inputs to the match function used in lab assignment 12, result in extend-match being called exactly twice. Also show the inputs for each of the two calls to extend-match. (The pattern matching code appears on the last page of this exam.) 6 pts A couple of possibilities: (match '(* a) '(b c a)) (extend-match '(* a) '(c a) '((b))) (extend-match '(* a) '(a) '((b c))) (match '(* a * b) '(x a y b)) (extend-match '(* a * b) '(a y b) '((x))) (extend-match '(* b) '(b) '((y) (x))) 1 pt for each input of each call 2 pts out of 6 if they only give a correct match call without the calls to extend-match -2 for off-by-one (i.e. for assuming that extend-match is called for each possible assignment to *, including the null assignment) Problem 6 (3 points, 15 minutes) One of the rule-based applications demonstrated in class used a version of the pattern-matching code, modified to return the * list instead of T for a successful match. One modification was necessary to do this; in match- with-*-list, the first cond

clause was changed from ((null pattern) (null object)) to ((null pattern) (if (null object) *-list nil)) In some cases, however, this modification will result in match returning a true value where it would previously have returned nil, or returning nil where it would previously have returned a true value. Give a general description of all patterns and objects for which the modified pattern matching code gives a different result from the unmodified code, that is, for which match with the modified code returns a true value where it would previously have returned nil, or returns nil where it would previously have returned a true value. 3 pts any pattern with no special pattern elements Pattern matching code from lab assignment 12 (defun match (pattern object) (match-with-*-list pattern object nil)) (defun match-with-*-list (pattern object *-list) (cond ((null pattern) (null object)) ((prev-ref? (first pattern)) (match-with-*-list (append (matched-value (first pattern) *-list) (rest pattern)) object *-list)) ((null object) (and (arb-sequence? (first pattern)) (match-with-*-list (rest pattern) object (add-*-entry *-list nil)))) ((arb-sequence? (first pattern)) (or (match-with-*-list (rest pattern) object (add-*-entry *-list nil)) (extend-match pattern (rest object) (add-*-entry *-list (first object))))) ((symbol? (first pattern)) (and (equal (first pattern) (first object)) (match-with-*-list (rest pattern) (rest object) *-list))) (T 'INVALID-PATTERN))) (defun extend-match (pattern object *-list) (cond ((match-with-*-list (rest pattern) object *-list)) ((null object) nil) (T (extend-match pattern (rest object) (extend-*-entry *-list (first object)))))) (defun arb-sequence? (pattern-elem) (equal pattern-elem '*)) (defun symbol? (pattern-elem) (and (atom pattern-elem) (not (arb-sequence? pattern-elem)) (not (prev-ref? pattern-elem)))) (defun prev-ref? (pattern-elem) (numberp pattern-elem)) (defun add-*-entry (*-list item) (if (null item) (cons item *-list) (cons (list item) *-list))) (defun extend-*-entry (*-list item) (cons (append (first *-list) (list item)) (rest *-list))) (defun matched-value (n *-list) (nth (- n 1) (reverse *-list)))