What I Graded On

I took off one or more points for, in decreasing importance:

- Incorrect algorithm
- No algorithm description
- Inefficient code (repeated identical calls to functions, unnecessary construction of intermediate data structures)
- Inexperienced code (failure to use appropriate built-in functions and data structures)
- Bugs

My Algorithm

Since the most important goals that are affected differently determine the result, my algorithm loops down the goals from most to least important, and stops as soon as a difference is found.

For each set of equivalently important goals,

- Compare the plans on that goal set (see below).
- If the result is BETTER, WORSE or MIXED, stop and return that result

If no difference is found for any set, return SAME.

To compare plans for a set of equivalently important goals:

- Find a goal where the plans differ. Call this difference D.
    - If there is no such goal, return “no difference.”
- Find a subsequent goal where the plans differ differently.
    - If there is no such goal, return D.
    - Otherwise return MIXED.

No intermediate data structures need to be constructed to do this. My implementation in Lisp (below) uses one loop through a set of goals, with a variable to hold the first difference found. As soon as a second different difference is found, it can return a result. I make extensive use of `some` to communicate the “find first difference” aspect of the algorithm.

My Implementation

```
;;; Some test cases, with all possible comparison results,
;;; including unknown plans. Plan P3 added to get MIXED result.

(define-test compare-plans
  (let ((goals '((A B) (C) (D E F) (G H)))
        (plans '((P1 (+ B E F H) (- C D G))
                 (P2 (+ C D F) (- E H))
                 (P3 (+ A D) (- C F))
                 )))
    (assert-equal :better (compare-plans 'p1 'p2 goals plans))
    (assert-equal :worse (compare-plans 'p2 'p1 goals plans))
    (assert-equal :same (compare-plans 'p1 'p1 goals plans))
    (assert-equal :mixed (compare-plans 'p3 'p1 goals plans))
    (assert-equal :mixed (compare-plans 'p1 'p3 goals plans))
    (assert-equal :better (compare-plans 'p3 'p2 goals plans))
    (assert-equal :same (compare-plans 'p4 'p5 goals plans))
    ))

;;; Either some difference is found, or the result is SAME.
(defun compare-plans (p1 p2 &optional goal-data plan-data)
  (or (some #'(lambda (goals)
                (difference-on-goals p1 p2 goals plan-data))
            goal-data)
      :same))

;;; Loop over goals, get the first difference, exit loop if a
;;; second non-NIL different difference is found.
(defun difference-on-goals (p1 p2 goals plan-data)
  (let ((first-diff nil))
    (or (some #'(lambda (goal)
                  (let ((diff
                         (difference-on-goal p1 p2 goal plan-data)))
                    (cond ((null diff) nil)
                          ((null first-diff)
                           (setq first-diff diff)
                           nil)
                          ((eql diff first-diff) nil)
                          (t :mixed))))
              goals)
        first-diff)))

;;; Return whether plan p1 is better, worse or neither for a goal
;;; than p2.
(defun difference-on-goal (p1 p2 goal plan-data)
  (compare-effects (plan-goal-effect p1 goal plan-data)
                   (plan-goal-effect p2 goal plan-data)))

(defun compare-effects (e1 e2)
  (cond ((eql e1 e2) nil)
        ((or (eql e1 '+) (and (null e1) (eql e2 '-)))
         :better)
        (t :worse)))

;;; Return +, - or nil, for how a plan affects a goal.
(defun plan-goal-effect (p goal plan-data)
  (car (find goal (cdr (assoc p plan-data)) :test #'member)))
```