General Instructions

Mutation, such as `set!` or `set-mcar!`, is generally poor style, so give at most a 3 to solutions using mutation.

Problem 1

Here is a sample solution:

```
(define (racketlist->mupllist xs)
  (if (null? xs)
      (aunit)
      (apair (car xs) (racketlist->mupllist (cdr xs)))))

(define (mupllist->racketlist xs)
  (if (aunit? xs)
      null
      (cons (apair-e1 xs) (mupllist->racketlist (apair-e2 xs)))))
```

It is unlikely that there are many ways to make solutions much longer or more complicated while having style as good as the sample solution, so mostly follow general guidelines.

Solutions using higher-order functions, such as `foldl` or `foldr` from Racket's standard library, can get a 5 if they are clear and concise.

Some students may have misinterpreted the problem as requiring recurring into nested lists (even though the problem said to presume the list contents were MUPL values and therefore shouldn't be changed). This can lead to more complicated and unnecessary code, but from a style perspective, there is no need to give significant penalties if this is the only complication, so otherwise good solutions would probably deserve a 4.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 2

For problem 2, we will break the assessment down into more manageable pieces with separate grades for each piece.

Problem 2: Overall Structure

Here evaluate if their solution has the right overall structure of having one case for each kind of MUPL expression. Some helper functions outside the big cond expression are okay although the sample solution does not have any.

```
(define (eval-under-env e env)
  (cond [(var? e) ...]
        [(int? e) ...]
        [(add? e) ...]
        [(ifgreater? e) ...]
        [(fun? e) ...]
        [(call? e) ...]
        [(mlet? e) ...]
        [(apair? e) ...]
        [(fst? e) ...]
        [(snd? e) ...]
        [(aunit? e) ...]
        [(isaunit? e) ...]
        [(closure? e) ...]
        [#t ...]))
```

Give a 3 or a 4 to solutions that do not follow this structure depending on how clear the overall structure is.

The order of cases does not matter.

It is fine not to have the last `#t` case since we are assuming the input is a legal AST.

If other cases are missing but the structure is right, you can give a 5 here and take off in questions below about those cases.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 2: var/int/aunit/closure/fun

Here assess the 5 very short cases:

```
[(var? e) (envlookup env (var-string e))]

[(int? e) e]

[(aunit? e) e]

[(closure? e) e]

[(fun? e) (closure env e)]
```

The three cases for values should return the entire expression. Give a 4 for unnecessary copying of the value, like (`int (int-num e)`). Give at most a 3 and possibly a 2 for cases much more complicated than that. (If most cases are great, but, for example, only the `aunit` case is worse, then that can average to a 4 or a 3.)

The `var?` case was given and should not be modified -- give at most a 3 to this question if it is not exactly this call to `envlookup`.

The `fun?` case is the most interesting here. Give at most a 4 if it is more complicated than (`closure env e`) and give at most a 3 for this problem if a function is returned instead of a closure or they do not return the entire function.

If the closure case is missing, do not penalize the solution because the auto-grader already did and it is disputable whether this case is needed (though the assignment specifically indicated to have it return the entire expression like for other values).

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 2: add and ifgreater

Here assess the cases for `add` and `ifgreater`.

```
          [(add? e)
          (let ([v1 (eval-under-env (add-e1 e) env)]
                [v2 (eval-under-env (add-e2 e) env)])
            (if (and (int? v1)
                     (int? v2))
                (int (+ (int-num v1)
                        (int-num v2)))
                (error "MUPL addition applied to non-number")))]

          [(ifgreater? e)
           (let ([v1 (eval-under-env (ifgreater-e1 e) env)]
                 [v2 (eval-under-env (ifgreater-e2 e) env)])
             (if (and (int? v1)
                      (int? v2))
                 (if (> (int-num v1) (int-num v2))
                     (eval-under-env (ifgreater-e3 e) env)
                     (eval-under-env (ifgreater-e4 e) env))
                 (error "MUPL ifgreater applied to non-number")))]
```

The `add` case was provided and should not be modified. We include it here because it is similar and an interesting contrast to the `ifgreater` case. Give at most a 3 if `add` was changed unless it was somehow changed well to use helper functions used in other cases.

For `ifgreater`, there are various fine ways to use let-expressions, but it is important that:

- `e1` and `e2` are evaluated exactly once
- One of `e3` and `e4` are evaluated once and the other is evaluated zero times

Give at most a 3 if the expressions are not evaluated the correct number of times.

Give at most a 4 if the case does not check correctly that the results of `e1` and `e2` are MUPL numbers.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 2: apair

Here we assess the `apair` case:

```
1           [(apair? e)
2            (apair (eval-under-env (apair-e1 e) env)
3                   (eval-under-env (apair-e2 e) env))]
```

Using a let-expression to bind the results of the recursive calls is okay, but as the sample solution shows, it is not necessary.

Give at most a 4 if there is unnecessary error-checking: It is legal to make a MUPL pair out of any two MUPL values, so there is no dynamic type-checking in this case.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 2: fst/snd

Here we assess the `fst` and `snd` cases together because they are very similar.

```
         [(fst? e)
          (let ([pr (eval-under-env (fst-e e) env)])
            (if (apair? pr)
                (apair-e1 pr)
                (error "MUPL fst applied to non-apair")))]
         [(snd? e)
          (let ([pr (eval-under-env (snd-e e) env)])
            (if (apair? pr)
                (apair-e2 pr)
                (error "MUPL snd applied to non-pair")))]

```

Deduct at least one point (i.e., from a 5 to a 4 or from a 4 to a 3, etc.) if the interpreter is called more than once on the subexpression. Getting this right requires some form of `let` or local `define`. (Deduct only one point total even if the `fst` and `snd` cases make the same mistake, not one point for each case.)

Deduct at least one point if the case does not check that the recursive result is a MUPL pair (using `apair?`). (Deduct only one point total even if the `fst` and `snd` cases make the same mistake.)

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 2: isaunit

Here we assess the `isaunit` case:

```
         [(isaunit? e)
          (let ([v (eval-under-env (isaunit-e e) env)])
            (if (aunit? v) (int 1) (int 0)))]

```

Although the sample solution uses a let-expression, it is not needed in this case.

Deduct a point if any dynamic type-checking is done. This is not correct because `isaunit?` can be used with any kind of MUPL value.

To give a sense of alternate solutions, this would also be fine:

`(int (if (aunit? (eval-under-env (isaunit-e e) env)) 1 0))`

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 2: mlet

Here we assess the mlet case:

```
1          [(mlet? e)
2           (let* ([v (eval-under-env (mlet-e e) env)]
3                  [newenv (cons (cons (mlet-var e) v) env)])
4             (eval-under-env (mlet-body e) newenv))]
5
```

A let-expression is not needed here, but do make sure there are exactly two recursive calls to `eval-under-env`.

Creating the new environment should not be complicated: make sure it is clear that one pair (of (`mlet-var e`) and the result of evaluating (`mlet-e e`) under `env`) is consed onto `env`.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 2: call

Here we assess the most interesting case, the `call` case:

```
1          [(call? e)
2           (let ([cl  (eval-under-env (call-funexp e) env)]
3                 [arg (eval-under-env (call-actual e) env)])
4             (if (closure? cl)
5                 (let* ([fn (closure-fun cl)]
6                        [bodyenv (cons (cons (fun-formal fn) arg)
7                                       (closure-env cl))]
8                        [bodyenv (if (fun-nameopt fn)
9                                     (cons (cons (fun-nameopt fn) cl)
10                                         bodyenv)
11                                    bodyenv)])
12                  (eval-under-env (fun-body fn) bodyenv))
13                (error "MUPL function call with nonfunction")))]
14
```

Deduct a point if there is not a clear check that the result of recursively evaluating (`call-funexp e`) is a closure.

Deduct at least a point if there are not clearly exactly three total recursive calls to `eval-under-env`. (It is fine if more calls appear in the code, for example separate calls for when the function being called has a name for recursion or not, but exactly three should be evaluated any time this case is evaluated.)

Deduct a point if the code always adds the `fun-nameopt` field to the environment even when it is `#f`. (This may likely work due to Racket's dynamic typing, but it is poor style to have non-strings in the environment.)

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 3a

Here is a sample solution:

```
(define (ifaunit e1 e2 e3) (ifgreater (isaunit e1) (int 0) e2 e3))
```

Give a 3 or a 4 for more complicated solutions. Give at most a 3 if the solution uses any Racket constructs, like `if`, that are not needed here.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 3b

Here is a sample solution:

```
(define (mlet* bs e2)
  (cond [(null? bs) e2]
        [#t (mlet (car (car bs)) (cdr (car bs))
                  (mlet* (cdr bs) e2))]))
```

Give at most a 3 for a solution that is not clearly using recursion to create nested let-expressions. However, a solution using a library function like `foldr` to do the recursion is great and can earn a 5.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 3c

Here is a sample solution:

```
(define (ifeq e1 e2 e3 e4)
  (mlet "_x" e1
        (mlet "_y" e2
              (ifgreater (var "_x") (var "_y")
                         e4
                         (ifgreater (var "_y") (var "_x")
                                    e4
                                    e3)))))
```

Deduct a point for a solution that does not clearly use two `ifgreater` expressions.

Deduct a point for a solution that does not use mlet to avoid repeated computation.

Deduct a point for a solution that uses unnecessary Racket computations, such as a Racket conditional.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 4a:

Here is a sample solution:

```
(fun "map" "f"
     (fun #f "xs"
          (ifaunit (var "xs")
                   (aunit)
                   (apair (call (var "f") (fst (var "xs")))
                          (call (call (var "map") (var "f"))
                                (snd (var "xs")))))))))
```

Another fine, arguably better, approach is to give a name to the inner function (e.g., g in place of `#f`) and then replace (`call` (`var` "`map`") (`var` "`f`")) with a call to the inner function (e.g., (`var` "`g`")).

Generally check that the solution looks like a MUPL program implementing the `map` function we have now seen many times.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Problem 4b

Here is a sample solution:

```
(define mupl-mapAddN
  (mlet "map" mupl-map
        (fun #f "x"
             (call (var "map") (fun #f "y"
                                    (add (var "x") (var "y")))))))
```

There are not too many ways to do this problem: We need a call to the `map` function defined in 4a with an appropriate function.

It is okay (and can earn a 5) if the solution gives names for recursion to functions that do not need them (i.e., replace the uses of `#f` above with strings).

It is okay if the solution uses more occurrences of mlet even though they are not needed.

Remember that you are grading on general style, not how close to the sample solution a student solution is. It is perfectly fine for a solution to be significantly different from the sample, as long as it has good style.

Challenge problem

You do not need to assess the challenge problem, but you are welcome to provide any feedback for it here. Here is a sample solution for it (where have elided all the interpreter cases that are the same except they replace `eval-under-env` with `eval-under-env-c`):

```
(define (compute-free-vars e)
  (struct res (e fvs)) ; result type of f (could also use a pair)
    (define (f e)
      (cond [(var? e) (res e (set (var-string e)))]
            [(int? e) (res e (set))]
            [(add? e) (let ([r1 (f (add-e1 e))]
                            [r2 (f (add-e2 e))])
                        (res (add (res-e r1) (res-e r2))
                             (set-union (res-fvs r1) (res-fvs r2))))]
            [(ifgreater? e) (let ([r1 (f (ifgreater-e1 e))]
                                  [r2 (f (ifgreater-e2 e))]
                                  [r3 (f (ifgreater-e3 e))]
                                  [r4 (f (ifgreater-e4 e))])
                              (res (ifgreater (res-e r1) (res-e r2) (res-e
                                  r3)    (res-e r4))
                                  (set-union (res-fvs r1) (res-fvs r2)
                                     (res-fvs   r3) (res-fvs r4))))]
            [(fun? e) (let* ([r (f (fun-body e))]
                             [fvs (set-remove (res-fvs r) (fun-formal e))]
                             [fvs (if (fun-nameopt e)
                                      (set-remove fvs (fun-nameopt e))
                                      fvs)])
                        (res (fun-challenge (fun-nameopt e) (fun-formal e)

                                            (res-e r) fvs)
                             fvs))]
            [(call? e) (let ([r1 (f (call-funexp e))]
                             [r2 (f (call-actual e))])
                         (res (call (res-e r1) (res-e r2))
                              (set-union (res-fvs r1) (res-fvs r2))))]
            [(mlet? e) (let* ([r1 (f (mlet-e e))]
                              [r2 (f (mlet-body e))])
                          (res (mlet (mlet-var e) (res-e r1) (res-e r2))
                               (set-union (res-fvs r1) (set-remove (res-fvs
                                   r2)   (mlet-var e)))))]
            [(apair? e) (let ([r1 (f (apair-e1 e))]
                              [r2 (f (apair-e2 e))])
                           (res (apair (res-e r1) (res-e r2))
                              (set-union (res-fvs r1) (res-fvs r2))))]
            [(fst? e) (let ([r (f (fst-e e))])
                        (res (fst (res-e r))
                             (res-fvs r)))]
            [(snd? e) (let ([r (f (snd-e e))])
                        (res (snd (res-e r))
                             (res-fvs r)))]
            [(aunit? e) (res e (set))]
            [(isaunit? e) (let ([r (f (isaunit-e e))])
                            (res (isaunit (res-e r))
                                 (res-fvs r)))]))
    (res-fvs (f e)))

(define (eval-under-env-c e env)
  (cond
        [(fun-challenge? e)
         (closure (set-map (fun-challenge-freevars e)
                           (lambda (s) (cons s (envlookup env s))))
                  e)]
         ; call case uses fun-challenge as appropriate
         ; all other cases the same
        ...)

```

```
58  (define (eval-exp-c e)
59    (eval-under-env-c (compute-free-vars e) null))
```

Mark as completed