Homework 4 – Starting Mutation

A set of tests will be supplied to you, but you should still make tests to insure that your functions are performing the operations you anticipate.

This is an individual assignment.

For a full grade, you only need to complete **Part 1** and **Part 2.**

Part 3 **and** Part 4 are bonus problems for people to make up points on previous assignments. They are 25 points each. **Place a comment at the top of your submission which indicates which problems you have attempted. If this isn't explicitly stated, I will assume that you didn't do either of the bonus problems.**

Part 1 — Improving Assignment

Start with store-with.rkt. In the starting program, the representation of the store grows every time that a box's content is modified with `set-box!` Change the implementation of `set-box!` so that the old value of the box is dropped (i.e., replaced with the new value) instead of merely hidden by the outside-in search order of `fetch`.

Example:

```
  (test (interp (parse `{let {[b {box 1}]}
                          {begin
                           {set-box! b 2}
                           {unbox b}}})
                mt-env
                mt-store)
        (v*s (numV 2)
             (override-store (cell 1 (numV 2))
                             mt-store)))
```

Part 2 — Sequences

Generalize `begin` to allow one or more sub-expressions, instead of exactly two sub-expressions.

```
  <Exp> = ...
        | {begin <Exp>+}
```

The `+` on `<Exp>+` means "one or more" in the same way that `*` means "zero or more." (See the hint below about nonempty lists.)

Example:

```
  (test (interp (parse `{let {[b {box 1}]}
                          {begin
                            {set-box! b {+ 2 {unbox b}}}
                            {set-box! b {+ 3 {unbox b}}}
                            {set-box! b {+ 4 {unbox b}}}
                            {unbox b}}})
                mt-env
                mt-store)
        (v*s (numV 10)
             (override-store (cell 1 (numV 10))
                             mt-store)))
```

Hint: As you know, a *list* has two cases: `empty` and (`cons` *item list*). A *nonempty-list* has two different cases: (`cons` *item* `empty`) and (`cons` *item nonempty-list*). Although `type-case` don’t know about *nonempty-list*s, you can use `cond` to follow the shape of that definition.

Part 3 — Records (extra)

Extend the interpreter to support the construction of records with named fields, to support field selection from a record (as in record.rkt):

```
  <Exp> = ...
        | {record {<Sym> <Exp>}*}
        | {get <Exp> <Sym>}
```

Adding records means that the language now has four kinds of values: numbers, functions, boxes, and records. At run-time, an error may occur because a record is misused as a number or function, a number or function is supplied to `get`, or a record supplied to `get` does not have the named field, and so on. Your error message for the last case should include the words “no such field”, but beyond that constraint you can make up your own error messages.

Expressions within a `record` form should be evaluated when the `record` form itself is evaluated, and in the order that the expressions appear in the `record` form. For example,

```
  {let {[b {box 0}]}
    {let {[r {record {a {unbox b}}}]}
      {begin
```

```
        {set-box! b 1}
        {get r a}}}}
```

should produce `0`, not `1`, because `{unbox b}` is evaluated when the `record` expression is evaluated, not when the `get` expression is evaluated.

Note that you will not be able to use `map` to `interp` field values, since a store must be carried from one field's evaluation to the next. Instead, `interp`ing the field value will be more like `interp`ing a sequence of expressions for `begin`.

For homework purposes, we don't want to nail down the representation of a record value, because there are many choices. The examples below therefore use `interp-expr`, which you should define as a wrapper on `interp` that takes just an `Exp` and produces just an S-expression: an S-expression number if `interp` produces any number, the S-expression `` `function `` if `interp` produces a closure, the S-expression `` `box `` if `interp` produces a box, or the S-expression `` `record `` if `interp` produces a record value.

Examples:

```
  (test (interp-expr (parse `{+ 1 4}))
        `5)
  (test (interp-expr (parse `{record {a 10} {b {+ 1 2}}}))
        `record)
  (test (interp-expr (parse `{get {record {a 10} {b {+ 1 0}}}
b}))
        `1)
  (test/exn (interp-expr (parse `{get {record {a 10}} b}))
            "no such field")
  (test (interp-expr (parse `{get {record {r {record {z 0}}}}
r}))
        `record)
  (test (interp-expr (parse `{get {get {record {r {record {z
0}}}} r} z}))
        `0)
  (test (interp-expr (parse `{let {[b {box 0}]}
                               {let {[r {record {a {unbox
b}}}]}
                                 {begin
                                   {set-box! b 1}
                                   {get r a}}}}))
        `0)
```

Part 4 — Mutating Records (extra)

Add a `set` form that modifies the value of a record field imperatively (as opposed to functional update):

```
  <Exp> = ...
        | {set! <Exp> <Sym> <Exp>}
```

Evaluation of a `record` expression allocates a location for each of its fields. A `get` expression accesses from the record produced by the sub-expression the value in the location of the field named by the identifier. A `set!` form changes the value in the location for a field; the value of the second sub-expression in `set!` determines the field's new value, and that value is also the result of the `set!` expression.

Note that making record fields mutable has the same effect as forcing every field of a record to be a Curly box, where the box contain the proper value of the field. Internal to the interpreter implementation, you could use Curly boxes in your implementation of mutable records, or you could use addresses more directly. You should not use Plait boxes at all.

Examples:

```
  (test (interp-expr (parse `{let {[r {record {x 1}}]}
                               {get r x}}))
        `1)

  (test (interp-expr (parse `{let {[r {record {x 1}}]}
                               {begin
                                 {set! r x 5}
                                 {get r x}}}))
        `5)

  (test (interp-expr (parse `{let {[r {record {x 1}}]}
                               {let {[get-r {lambda {d} r}]}
                                 {begin
                                   {set! {get-r 0} x 6}
                                   {get {get-r 0} x}}}}))
        `6)
  (test (interp-expr (parse `{let {[g {lambda {r} {get r a}}]}
                               {let {[s {lambda {r} {lambda {v}
{set! r b v}}}]}
                                 {let {[r1 {record {a 0} {b
2}}]}
                                   {let {[r2 {record {a 3} {b
4}}]}
                                     {+ {get r1 b}
                                        {begin
                                          {{s r1} {g r2}}
                                          {+ {begin
                                               {{s r2} {g r1}}
                                               {get r1 b}}
                                             {get r2
b}}}}}}}}}))
        `5)
```