CS 5510 Homework 5

Due: Wednesday, September 30th, 2015 11:59pm

Start with `variable.rkt`.

Part 1 — Records with Store

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

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

Adding records means that the language now has three kinds of values: numbers, functions, 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”, otherwise you can make up your own error messages.

Expressions within a `record` form should be evaluated when the `record` form itself is evaluated. For example,

```
{let {[x 0]}
  {let {[r {record {a x}}]}
    {begin
     {set! x 1}
     {get r a}}}}
```

should produce 0, not 1, because `x` 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 `ExprC` and produces just an S-expression: an S-expression number if `interp` produces any number, the S-expression `` `function `` if `interp` produces a closure, 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)
```

Part 2 — Mutating Records

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

```
<Expr> = ...
       | {set <Expr> <Sym> <Expr>}
```

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.

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 {[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)
```

Part 3 — Records with a No-Such-Field Handler

Your interpreter to support a `record/handle` form, in addition to `record`, and an `error` form:

```
<Expr> = ...
       | {record/handle <Expr> {<Sym> <Expr>}*}
       | {error <String>}
```

A `record/handle` form creates a record, just like `record` would with the same fields. However, when a `get` operation on a `record/handle` result attempts to access a field that is not present in the record, the `<Expr>` provided after the `record/handle` keyword is evaluated to produce the result of the `get`. Meanwhile, the `error` form should raise an error with the given message string.

In particular,

```
{record {x 1} {y 2}}
```

is equivlent to

```
{record/handle {error "no such field"} {x 1} {y 2}}
```

and you should consider implementing `record` that way.

The handler expression for a `record/handle`-produced value is *not* used with `set`.

Examples:

```
(test (interp-expr (parse '{let {[r {record/handle 5 {x 1}}]}
                             {get r x}}))
      '1)

(test (interp-expr (parse '{let {[r {record/handle 5 {x 1}}]}
                             {get r y}}))
      '5)

(test/exn (interp-expr (parse '{let {[r {record/handle {error "ouch"} {x 1}}]}
                                 {get r y}}))
          "ouch")

(test (interp-expr (parse '{let {[r {record/handle {error "ouch"} {x 1}}]}
                             {get r x}}))
      '1)
```

Last update: Monday, September 28th, 2015

mflatt@cs.utah.edu