Part I

Records

Literal objects in JavaScript:

```
var o = { x : 1,  y : 1+1 }

o.x  ⇒ 1
o.y  ⇒ 2
```

Record Update

```
var o = { x : 1,  y : 1+1 }
o.x = 5

o.x  ⇒ 5
```

This kind of update involves ***state***

We'll look at state on a different day

Record Functional Update

```
var o = { x : 1,  y : 1+1 }
var p = (o.x = 5)

o.x  ⇒ 1
p.x  ⇒ 5
p.y  ⇒ 2
```

This approach is ***functional update***

We'll implement functional update today

Records

```
{ x : 1,  y : 1+1 }
```

```
{record {x 1}
        {y {+ 1 1}}}
```

Records

```
var o = { x : 1,  y : 1+1 }
....
```

```
{let {[o {record {x 1}
                 {y {+ 1 1}}}]}
  ....}
```

Records

```
o.x
```

```
{get o x}
```

Records

```
var o = { x : 1,  y : 1+1 }
o.x
```

```
{let {[o {record {x 1}
                 {y {+ 1 1}}}]}
  {get o x}}
```

Records

```
(o.x = 5)
```

```
{set o x 5}
```

Functional Record Update

```
{let {[r1 {record {a {+ 1 1}}
                  {b {+ 2 2}}}]}
  {let {[r2 {set r1 a 5}]}
    {+ {get r1 a}
       {get r2 a}}}}
```

⇒ 7

`set` creates a new record with the new field value

Part 2

Records

```
<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  <Sym>
          |  {lambda {<Sym>} <Expr>}
          |  {<Expr> <Expr>}
          |  {record {<Sym> <Expr>} ...}        NEW
          |  {get <Expr> <Sym>}                 NEW
          |  {set <Expr> <Sym> <Expr>}          NEW
```

Record Programs

```
{let {[r {record {x 5}
                 {y 2}}]}
  {get r x}}
```

$\Rightarrow$ `5`

Record Programs

```
{let {[r {record {x 5}
                 {y 2}}]}
  {get r y}}
```

$\Rightarrow$ `2`

Record Programs

```
{let {[r {record {x 5}
                 {y {+ 1 1}}}]}
  {get r y}}
```

$\Rightarrow$ 2

Record Programs

```
{let {[mk {lambda {v}
            {record {x {+ v 1}}
                    {y {+ v 2}}}}]}
  {get {mk 2} x}}
```

⇒ 3

Record Programs

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

$\Rightarrow$ `1`

Record Programs

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

⇒ … a record …

Record Programs

```
{set {record {x 1}
             {y 2}}
     x
     5}
```

⇒ ... a record with `x` as `5`...

Record Expressions & Values

```
(define-type ExprC
  ....
  [recordC (ns : (listof symbol))
           (args : (listof ExprC))]
  [getC (rec : ExprC)
        (n : symbol)]
  [setC (rec : ExprC)
        (n : symbol)
        (val : ExprC)])

(define-type Value
  [numV (n : number)]
  [closV (arg : symbol)
         (body : ExprC)
         (env : Env)]
  [recV (ns : (listof symbol))
        (vs : (listof Value))])
```

Part 3

Parsing Records

```
(define (parse [s : s-expression]) : ExprC
  (cond
   ....
   [(s-exp-match? '{record {SYMBOL ANY} ...} s)
    (recordC (map (lambda (l)
                    (s-exp->symbol
                     (first (s-exp->list l))))
                  (rest (s-exp->list s)))
             (map (lambda (l)
                    (parse
                     (second (s-exp->list l))))
                  (rest (s-exp->list s))))]
   ....))
```

`interp` for Records

```
(define (interp [a : ExprC] [env : Env]) : Value
  (type-case ExprC a
    ...
    [setC (r n v)
          (type-case Value (interp r env)
            [recV (ns vs)
                  (recV ns
                        (update n
                                (interp v env)
                                ns
                                vs))]
            [else (error 'interp "not a record")])]
    ...))
```