Part I
Records

Literal objects in JavaScript:

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

o.x ⇒ 1
o.y ⇒ 2
```
Record Update

Field update in JavaScript:

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

o.x = 5
o.x ⇒ 5
```

This kind of update involves *state*
Record Functional Update

Field update *different* from JavaScript:

```javascript
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 first for Curly
Records

{ $x : 1, \ y : 1+1 \}$

{\texttt{record} \ {\texttt{x} \ 1} \\
{\texttt{y} \ {\texttt{+} \ 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

```javascript
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

```latex
\{let \{[[r1 \{record \{a 2\}
          \{b 4\}\}\}\]
         \{let \{[[r2 \{set r1 a 5\}\]\}
          \{+ \{get r1 a\}
           \{get r2 a\}\}\}\]\}

\Rightarrow 7

set creates a new record with the new field value
```
Part 2
Records

\[ <\text{Exp}> ::= <\text{Number}> \]
\[ | \{ + <\text{Exp}> <\text{Exp}> \} \]
\[ | \{ * <\text{Exp}> <\text{Exp}> \} \]
\[ | <\text{Symbol}> \]
\[ | \{ \text{lambda} \{ <\text{Symbol}> \} <\text{Exp}> \} \]
\[ | \{ <\text{Exp}> <\text{Exp}> \} \]
\[ | \{ \text{record} \{ <\text{Symbol}> <\text{Exp}> \} \ldots \} \]
\[ | \{ \text{get} <\text{Exp}> <\text{Symbol}> \} \]
\[ | \{ \text{set} <\text{Exp}> <\text{Symbol}> <\text{Exp}> \} \]
Record Programs

\[
\begin{align*}
\{ \text{let} \{[[r \{\text{record} \{x \ 5\} \\
\quad \{y \ 2\}\}]]} \\
\quad \{\text{get} \ r \ x\}\} \\
\Rightarrow \ 5
\end{align*}
\]
Record Programs

\[
\{\text{let } \{\{r \{\text{record } \{x \ 5\} \\
\quad \{y \ 2\}\}\}\}\} \\{\text{get } r \ y\}\}\}
\Rightarrow \ 2
\]
Record Programs

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

⇒ 2
```
Record Programs

\[
\begin{align*}
&\text{let } \{ [mk \ {\lambda} \ {v} \\
&\text{record } \{ x \ {+} \ v \ 1 \}
\{ y \ {+} \ v \ 2 \} \} \} \} \\
&\text{get } \{ mk \ 2 \ x \} \\
\Rightarrow 3
\end{align*}
\]
Record Programs

```plaintext
{get {record {x 1} {y 2}} x} ⇒ 1
```
Record Programs

\[
\{ \text{record } \{ x \ 1 \} \\
\{ y \ 2 \} \} \\
\Rightarrow \ ... \ a \ record ... 
\]
Record Programs

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

⇒ ... a record with x as 5...
Record Expressions & Values

(define-type Exp

....

(recordE [ns : (Listof Symbol)]
  [args : (Listof Exp)])

(getE [rec : Exp]
  [n : Symbol])

(setE [rec : Exp]
  [n : Symbol]
  [val : Exp]))

(define-type Value

(numV [n : Number])

(closV [arg : Symbol]
  [body : Exp]
  [env : Env])

(recV [ns : (Listof Symbol)]
  [vs : (Listof Value)]))
Part 3
Parsing Records

(define (parse [s : S-Exp]) : Exp
  (cond
    ....
    [(s-exp-match? `{record {SYMBOL ANY} ...} s)
      (recordE (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 : Exp] [env : Env]) : Value
  (type-case Exp a
    ...
    [(setE 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")]]
    ...))

32-43
Updating a Record

(define (update [n : Symbol]  
    [v : Value]  
    [ns : (Listof Symbol)]  
    [vs : (Listof Value)]) : (Listof Value)

(cond
  [(empty? ns) (error 'interp "no such field")]
  [else (if (symbol=? n (first ns))
             (cons v (rest vs))
             (cons (first vs)
                   (cons (update n v (rest ns) (rest vs))))))])
Part 4
Imperative Record Update

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

o.x ⇒ 5
```

Creating a JavaScript object allocates memory for each of its fields

Field assignment updates memory
Imperative Record Update

```scheme
(let [[r1 {record {a (+ 1 1)}
            {b (+ 2 2)}}]]
  {begin
    {set! r1 a 5}
    {get r1 a}})
⇒ 5

Creating a record must allocate memory for each of its fields

Curly’s new `set!` modifies a field’s memory, instead of creating a new object
Records with Allocated Fields via Boxes

(define-type Value
    ...
    (recV [ns : (Listof Symbol)]
        [vs : (Listof (Boxof Value))])))
interp for Mutable Records

(define (interp [a : Exp] [env : Env]) : Value
  (type-case Exp a 
    ...
    [(recordE ns vs) 
      (recV ns 
        (map (lambda (v) (interp v env)) 
             vs))] 
    ...)))
interp for Mutable Records

(define (interp [a : Exp] [env : Env]) : Value
  (type-case Exp a
    ...
    [(recordE ns vs)
      (recV ns
        (map (lambda (v) (box (interp v env)))
             vs))]
    ...))
interp for Mutable Records

(define (interp [a : Exp] [env : Env]) : Value
  (type-case Exp a
    ...
    [(getE r n)
      (type-case Value (interp r env)
        [(recV ns vs)
          (find n ns vs)]
        [(else error 'interp "not a record")])])
  ...
)

find : (Symbol (Listof Symbol) (Listof Value)
  -> Value)
**interp** for Mutable Records

```scheme
(define (interp [a : Exp] [env : Env]) : Value
  (type-case Exp a
    ...
    [(getE r n)
      (type-case Value (interp r env)
        [(recV ns vs)
          (unbox (find n ns vs))]
        [(else error 'interp "not a record")])]
    ...
))

find : (Symbol (Listof Symbol) (Listof (Boxof Value))
    -> (Boxof Value))
```
interp for Mutable Records

(define (interp [a : Exp] [env : Env]) : Value
  (type-case Exp a
    ...
    [(setE r n v)
      (type-case Value (interp r env)
        [(recV ns vs)
          .... (find n ns vs) ....]
        [(else error 'interp "not a record")])]]
    ...
))

find : (Symbol (Listof Symbol) (Listof (Boxof Value)))
     -> (Boxof Value)
interp for Mutable Records

(define (interp [a : Exp] [env : Env]) : Value
  (type-case Exp a
    ...
    [(setE r n v)
     (type-case Value (interp r env)
       [(recV ns vs)
        (set-box! (find n ns vs) (interp v env))]
       [(else error 'interp "not a record")])]
    ...))

find : (Symbol (Listof Symbol) (Listof (Boxof Value)))
    -> (Boxof Value)
interp for Mutable Records

(define (interp [a : Exp] [env : Env]) : Value
  (type-case Exp a
    ...
    [(setE r n v)
      (type-case Value (interp r env)
        [(recV ns vs)
          (let ([f (interp v env)])
            (begin
              (set-box! (find n ns vs) f)
              f)]
          [(else error 'interp "not a record")])]
    ...
  ))
interp for Mutable Records

(define (interp [a : Exp] [env : Env]) : Value
  (type-case Exp a
    ...
    [(recordE ns vs)
      (recV ns
        (map (lambda (v) (box (interp v env)))
             vs))]
    ...))
interp for Mutable Records

(define (interp [a : Exp] [env : Env]) : Value
  (type-case Exp a
    ...
    [(recordE
      (recV ns
       (map (lambda (v) (box (interp v env)))
            vs))]
    ...
  ))

Won't work with a store!
Part 5
API Terminology

**Imperative update** = **Mutable datatype**

```scheme
> (define ht
    (make-hash (list (values 'a 1)
                    (values 'b 2))))

> (hash-ref ht 'a)
(some 1)

> (hash-set! ht 'a 3)

> (hash-ref ht 'a)
(some 3)
```
API Terminology

**Functional update** = **Persistent datatype**

```
> (define ht
  (hash (list (values 'a 1)
           (values 'b 2))))

> (hash-ref ht 'a)
(some 1)

> (define ht2 (hash-set ht 'a 3))

> (hash-ref ht2 'a)
(some 3)

> (hash-ref ht 'a)
(some 1)
```