Part I

Records

Literal objects in JavaScript:

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

o.x  ⇒ 1
o.y  ⇒ 2
```

Record Update

Field update in JavaScript:

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

o.x = 5
o.x  ⇒ 5
```

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

Functional Record Update

Field update *different* from 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 Moe

Records

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

Records

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

Records

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

Records

```
o with (x = 5)
```

Functional Record Update

```
let r1 = { a: 2,
           b: 4 }:
  let r2 = r1 with (a = 5):
    r1.a + r2.a
```

⇒ 7

`obj with (field = expr)` creates a new record with the new field value

Part 2

Records

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  { <Symbol>: <Exp>, ... }             NEW
         |  <Exp>.<Symbol>                       NEW
         |  <Exp> with (<Symbol> = <Exp>)        NEW
```

Record Programs

```
let r = { x: 5,
          y: 2 }:
  r.x
```

⇒ 5

Record Programs

```
let r = { x: 5,
          y: 2 }:
  r.y
```

⇒ 2

Record Programs

```
let r = { x: 5,
          y: 1 + 1 }:
  r.y
```

⇒ 2

Record Programs

```
let mk = (fun (v):
            { x: v + 1,
              y: v + 2 }):
  mk(2).x
```

⇒ 3

Record Programs

```
{ x: 1,
  y: 2 }.x
```

⇒ 1

Record Programs

```
{ x: 1,
  y: 2 }
```

⇒ ... a record ...

Record Programs

```
{ x: 1,
  y: 2 } with (x = 5)
```

⇒ ... a record with `x` as 5...

Record Expressions & Values

```
type Exp
....
| recordE(ns :: Listof(Symbol),
          args :: Listof(Exp))
| getE(rec :: Exp,
       n :: Symbol)
| setE(rec :: Exp,
       n :: Symbol,
       val :: Exp)

type Value
....
| recV(ns :: Listof(Symbol),
       vs :: Listof(Value))
```

Part 3

Parsing Records

```
fun parse(s :: Syntax) :: Exp:
  match
  ....
  | '{ $field: $expr, ... }':
      recordE(map(syntax_to_symbol,
                  syntax_to_list('[$field, ...]')),
              map(parse,
                  syntax_to_list('[$expr, ...]')))
  | ....
```

interp for Records

```
fun interp(a :: Exp, env :: Env) :: Value:
  ....
  | setE(r, n, v):
      match interp(r, env)
      | recV(ns, vs):
          recV(ns,
               update(n,
                      interp(v, env),
                      ns,
                      vs))
      | else(error, #'interp, "not a record")()
  ....
```

Functional Record Update

```
fun update(n :: Symbol,
           v :: Value,
           ns :: Listof(Symbol),
           vs :: Listof(Value)) :: Listof(Value):
  match ns
  | []: error(#'interp, "no such field")
  | cons(ns_n, ns_rst): if n == ns_n
                        | cons(v, rest(vs))
                        | cons(first(vs),
                               update(n, v, ns_rst, 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

```
let r1 = { a: 1 + 1,
           b: 2 + 2 }:
  begin:
    r1.a := 5
    r1.a
```

⇒ 5

Creating a record must allocate memory for each of its fields

`obj.field := rhs` modifies a field's memory, instead of creating a new record

Records with Allocated Fields via Boxes

```
type Value
....
| recV(ns :: Listof(Symbol),
       vs :: Listof(Boxof(Value)))
```

`interp` for Mutable Records

```
fun interp(a :: Exp, env :: Env) :: Value:
  ....
  | recordE(ns, vs):
      recV(ns,
           map(fun (v): interp(v, env),
               vs))
  ....
```

interp for Mutable Records

```
fun interp(a :: Exp, env :: Env) :: Value:
  ....
  | recordE(ns, vs):
      recV(ns,
           map(fun (v): box(interp(v, env)),
               vs))
  ....
```

interp for Mutable Records

```
fun interp(a :: Exp, env :: Env) :: Value:
  ....
  | getE(r, n):
      match 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

```
fun interp(a :: Exp, env :: Env) :: Value:
  ....
  | getE(r, n):
      match 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

```
fun interp(a :: Exp, env :: Env) :: Value:
  ....
  | setE(r, n, v):
      match 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

```
fun interp(a :: Exp, env :: Env) :: Value:
  ....
  | setE(r, n, v):
      match 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

```
fun interp(a :: Exp, env :: Env) :: Value:
  ....
  | setE(r, n, v):
      match interp(r, env)
      | recV(ns, vs):
          let f = interp(v, env):
            set_box(find(n, ns, vs), f)
            f
      | else(error, #'interp, "not a record")()
  ....
```

interp for Mutable Records

```
fun interp(a :: Exp, env :: Env) :: Value:
  ....
  | recordE(ns, vs):
      recV(ns,
           map(fun (v): box(interp(v, env)),
               vs))
  ....
```

interp for Mutable Records

```
fun interp(a :: Exp, env :: Env) :: Value:
  ....
  | recordE(r
      recV(ns,
           map(fun (v): box(interp(v, env)),
               vs))
  ....
```

Won't work with a store!

Part 5

API Terminology

Imperative update = ***Mutable datatype***

```
> def ht:
    MutableMap{ #'a: 1,
                #'b: 2 }

> map_get(ht, #'a)
some(1)

> map_set(ht, #'a, 3)

> map_get(ht, #'a)
some(3)
```

API Terminology

Functional update = ***Persistent datatype***

```
> def ht:
    { #'a: 1,
      #'b: 2 }

> map_get(ht, #'a)
some(1)

> def ht2 = map_update(ht, #'a, 3)

> map_get(ht2, #'a)
some(3)

> map_get(ht, #'a)
some(1)
```