




# Records — Part I

# Records

```
struct Posn {  
    int x, y;  
}
```

```
Posn p = new Posn();  
p.x = 0;  
p.y = 0;
```

# Records

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

# Record Programs

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

⇒ 5

# Record Programs

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

⇒ 2

# Record Programs

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

⇒ 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}
```

⇒ 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))])
```

## Records — Part 2

# 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")])]
    ...))
```

# Records — Part 3

# 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

This is called ***functional update***



# Imperative Record Update

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

⇒ 10

assuming that **set** mutates a record's field

... like a box

This is called ***imperative update***

# Record Expressions & Values

```
(define-type Value
  ....
  [recV (ns : (listof symbol))
        (vs : (listof (boxof Value)))])
```

## interp for Mutable 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)
          (let ([bx (find n ns vs)])
            (begin
              (set-box! bx (interp v env))
              (recV ns
                vs))))]
        [else (error 'interp "not a record")])]
    ...))
```

# Encoding — Part I

# Binding Constructs

```
{let {[x 5]}  
    {+ x 6}}
```

```
{let {[f {lambda {x}  
          {+ x 6}}]}  
    {f 5}}
```

# Encoding let

These programs are the same:

```
{let { [x 5] }  
    {+ x 6}}
```

```
{{lambda {x}  
    {+ x 6}}  
5}
```

# Encoding let

These programs are the same:

```
{let {[x 5]}  
  body}
```

```
{{lambda {x}  
  body}  
5}
```

# Encoding let

These programs are the same:

```
{let { [x rhs] }  
  body}
```

```
{ {lambda {x}  
  body}  
  rhs}
```



# Encoding let

These programs are the same:

```
{let {[name rhs]}  
  body}
```

```
{{lambda {name}  
  body}  
 rhs}
```

```
(test (parse '{let {[x 5]} {+ x 6}})  
      (appC (lamC 'x (plusC (idC 'x) (numC 6)))  
            (numC 5)))
```

# Encoding Multiple Arguments

```
{let {[f {lambda {x y}
          {+ x y}}]}
      {f 1 2}}
```

```
{let {[f {lambda {x}
          {lambda {y}
            {+ x y}}}}]}
      {{f 1} 2}}
```

# Encoding Multiple Arguments

```
{let {[f {lambda {x y}
          body}]}
      {f 1 2}}
```

```
{let {[f {lambda {x}
          {lambda {y}
            body}}]}
      {{f 1} 2}}
```

This transformation is called **currying**

# Encoding — Part 2

# Encoding if

```
{if tst  
  thn  
  els}
```

# Encoding if

```
{if* tst  
  {lambda {d} thn}  
  {lambda {d} els}}
```

# Encoding if

```
{ {if* tst
      {lambda {d} thn}
      {lambda {d} els}}
0 }
```

```
true def = {lambda {x} {lambda {y} x}}
false def = {lambda {x} {lambda {y} y}}
```

```
{ {{{tst
      {lambda {d} thn}}
      {lambda {d} els}}}}
0 }
```

# Encoding Pairs

```
{cons 1 empty}
```



# Encoding Pairs

```
{cons 1 0}
```

# Encoding Pairs

```
{cons f r}
```

# Encoding Pairs

`{lambda ... f r}`

# Encoding Pairs

```
{lambda {sel} {{sel f} r}}
```

```
cons def = {lambda {x}  
           {lambda {y}  
             {lambda {sel} {{sel x} y}}}}
```

```
first def = {lambda {p} {p true}}
```

```
rest def = {lambda {p} {p false}}
```

```
{first {{cons 1} 0}}  
⇒ {first {lambda {sel} {{sel 1} 0}}}  
⇒ {{lambda {sel} {{sel 1} 0}} true}  
⇒ {{true 1} 0}  
= {{{lambda {x} {lambda {y} x}} 1} 0}  
⇒ {{lambda {y} 1} 0}  
⇒ 1
```

# Encoding — Part 3

# $\lambda$ -Calculus Grammar

$\langle \text{Expr} \rangle ::= \langle \text{Sym} \rangle$   
|  $\{ \langle \text{Expr} \rangle \langle \text{Expr} \rangle \}$   
|  $\{ \text{lambda } \{ \langle \text{Sym} \rangle \} \langle \text{Expr} \rangle \}$

# $\lambda$ -Calculus Grammar

$\langle \text{Expr} \rangle ::= \langle \text{Sym} \rangle$   
|  $\{ \langle \text{Expr} \rangle \langle \text{Expr} \rangle \}$   
|  $(\lambda (\langle \text{Sym} \rangle) \langle \text{Expr} \rangle)$

$\text{true} \stackrel{\text{def}}{=} (\lambda (\mathbf{x}) (\lambda (\mathbf{y}) \mathbf{x}))$

$\text{false} \stackrel{\text{def}}{=} (\lambda (\mathbf{x}) (\lambda (\mathbf{y}) \mathbf{y}))$

# Encoding — Part 4



# Encoding Numbers

`zero` <sup>def</sup> `(λ (x) (λ (y) y))`

# Encoding Numbers

**zero**  $\stackrel{\text{def}}{=} (\lambda (f) (\lambda (y) y))$

**one**  $\stackrel{\text{def}}{=} (\lambda (f) (\lambda (y) \{f y\}))$

**two**  $\stackrel{\text{def}}{=} (\lambda (f) (\lambda (y) \{f \{f y\}\}))$

**three**  $\stackrel{\text{def}}{=} (\lambda (f) (\lambda (y) \{f \{f \{f y\}\}\}))$

**N**  $\stackrel{\text{def}}{=} (\lambda (f) (\lambda (y) \{f_1 \dots \{f_N y\}\}))$

# Incrementing a Number

```
add1 def = (λ (n)  
            . . . )
```

# Incrementing a Number

```
add1 def = (λ (n)  
            (λ (f)  
              (λ (x) . . .)))
```

# Incrementing a Number

```
add1 def = (λ (n)
             (λ (f)
              (λ (x) ... {{n f} x} ...)))
```

# Incrementing a Number

```
add1 def = (λ (n)
            (λ (f)
              (λ (x) {f {{n f} x}})))
```

```
(add1 zero)
⇒ (λ (f)
   (λ (x) {f {{zero f} x}}))
= (λ (f)
   (λ (x) {f {{(λ (f) (λ (x) x)) f} x}}))
⇒ (λ (f)
   (λ (x) {f x}))
= one
```

# Adding Numbers

`add2`  $\stackrel{\text{def}}{=} (\lambda (n) \{ \text{add1} \{ \text{add1} n \} \})$

`add3`  $\stackrel{\text{def}}{=} (\lambda (n) \{ \text{add1} \{ \text{add1} \{ \text{add1} n \} \} \})$

`add`  $\stackrel{\text{def}}{=} (\lambda (n) (\lambda (m) \{ \text{add1}_1 \dots \{ \text{add1}_m n \} \}))$

# Adding Numbers

`add2`  $\stackrel{\text{def}}{=} (\lambda (n) \{ \text{add1} \{ \text{add1} n \} \})$

`add3`  $\stackrel{\text{def}}{=} (\lambda (n) \{ \text{add1} \{ \text{add1} \{ \text{add1} n \} \} \})$

`add`  $\stackrel{\text{def}}{=} (\lambda (n) (\lambda (m) \{ \{ m \text{ add1} \} n \}))$

... because a number  $m$  applies some function  $m$  times to an argument

`{ {add one} two }`  
 $\Rightarrow$  `{ {two add1} one }`  
 $\Rightarrow$  `{ add1 {add1 one} }`  
 $\Rightarrow$  `three`



# Multiplying Numbers

```
mult def = (λ (n) (λ (m) {{add n} |  
    . . .  
    {{add n} m zero}}))
```

# Multiplying Numbers

```
mult def = (λ (n) (λ (m) {{m {add n}} zero}))
```

... because `{add n}` is a function that adds  $n$  to any number

... and a number  $m$  applies some function  $m$  times to an argument

# Testing for Zero

```
iszero def = (λ (n) ... true ... false ...)
```

# Testing for Zero

```
iszero def = (λ (n) {{n (λ (x) false)}  
                true}))
```

because applying  $(\lambda (x) \text{ false})$  zero times to **true** produces **true**, and applying it any other number of times produces **false**

```
{iszero zero}  
⇒ {{zero (λ (x) false)} true}  
⇒ true
```

# Testing for Zero

```
iszero def = (λ (n) {{n (λ (x) false)}  
                true}))
```

because applying  $(\lambda (x) \text{ false})$  zero times to **true** produces **true**, and applying it any other number of times produces **false**

```
{iszero one}  
⇒ {{one (λ (x) false)} true}  
⇒ {(λ (x) false) true}  
⇒ false
```

# Decrementing a Number

```
sub1 def ≡ (λ (n)  
            (λ (f)  
              (λ (x) ...)))
```

# Decrementing a Number

```
sub1 def = (λ (n)  
            (λ (f)  
              (λ (x) ... {{n f} x} ...)))
```

Too late! No way to undo a call to **f**

# Decrementing a Number

```
... {{cons zero} one}  
... {{cons one} two}  
... {{cons two} three}  
...  
... {{cons n-1} n}
```



# Decrementing a Number

```
shift def = (λ (p)  
            {{cons {rest p}} {add1 {rest p}}})
```

```
sub1 def = (λ (n)  
           {first  
            {{n shift} {{cons zero} zero}}})
```

And then subtraction is obvious...

# Encodings

Using the minimal  $\lambda$ -calculus language we get

- ✓ functions
- ✓ local binding
- ✓ booleans
- ✓ numbers

# Encoding — Part 5

# Factorial

```
(local [(define fac
          (lambda (n)
            (if (zero? n)
                1
                (* n (fac (- n 1))))))]
  (fac 10))
```

`local` binds both in the body expression and in the binding expression

# Factorial

```
(letrec ([fac  
         (lambda (n)  
           (if (zero? n)  
               1  
               (* n (fac (- n 1))))))] )  
(fac 10))
```

`letrec` has the shape of `let` but the binding structure of `local`

# Factorial

```
(let ([fac  
      (lambda (n)  
        (if (zero? n)  
            1  
            (* n (fac (- n 1))))))] )  
(fac 10))
```

Doesn't work, because `let` binds `fac` only in the body

Still, at the point that we call `fac`, obviously we have a binding for `fac`...

... so pass it as an argument!

# Factorial

```
(let ([facX  
      (lambda (facX n)  
        (if (zero? n)  
            1  
            (* n (fac (- n 1))))))] )  
(facX facX 10))
```

# Factorial

```
(let ([facX  
      (lambda (facX n)  
        (if (zero? n)  
            1  
            (* n (facX facX (- n 1))))))] )  
(facX facX 10))
```

Wrap this to get `fac` back...



# Factorial

```
(let ([fac
      (lambda (n)
        (let ([facX
              (lambda (facX n)
                (if (zero? n)
                    1
                    (* n (facX facX (- n 1))))))]
          (facX facX n)))]])
  (fac 10))
```

Try this in the **HtDP Intermediate with Lambda** language, click **Step**

But the language we implement has only single-argument functions...

# Encoding — Part 6

# Factorial

```
(let ([fac
      (lambda (n)
        (let ([facX
              (lambda (facX)
                (lambda (n)
                  (if (zero? n)
                      1
                      (* n ((facX facX) (- n 1))))))]
          ((facX facX) n))))])
  (fac 10))
```

Simplify: `(lambda (n) (let ([f ...]) ((f f) n)))`  
 $\Rightarrow$  `(let ([f ...]) (f f))...`

# Factorial

```
(let ([fac
      (let ([facX
            (lambda (facX)
              (lambda (n)
                (if (zero? n)
                    1
                    (* n ((facX facX) (- n 1))))))]
          (facX facX))])
  (fac 10))
```

# Factorial

```
(let ([fac
      (let ([facX
            (lambda (facX)
              ; Almost looks like original fac:
              (lambda (n)
                (if (zero? n)
                    1
                    (* n ((facX facX) (- n 1))))))]
          (facX facX)))]
    (fac 10))
```

More like original: introduce a local binding for  
(facX facX)...

# Factorial

```
(let ([fac
      (let ([facX
            (lambda (facX)
              (let ([fac (facX facX)])
                ; Exactly like original fac:
                (lambda (n)
                  (if (zero? n)
                      1
                      (* n (fac (- n 1))))))]
                (facX facX)))]
      (fac 10)))
```

**Oops!** — this is an infinite loop

We used to evaluate `(facX facX)` only when `n` is non-zero

Delay `(facX facX)`...

# Factorial

```
(let ([fac
      (let ([facX
            (lambda (facX)
              (let ([fac (lambda (x)
                          ((facX facX) x))])
                ; Exactly like original fac:
                (lambda (n)
                  (if (zero? n)
                      1
                      (* n (fac (- n 1)))))))]
          (facX facX))]
      (fac 10)))
```

Now, what about **fib**, **sum**, etc.?

Abstract over the **fac**-specific part...

# Make-Recursive and Factorial

```
(define (mk-rec body-proc)
  (let ([fX
        (lambda (fX)
          (let ([f (lambda (x)
                    ((fX fX) x))])
            (body-proc f)))]])
    (fX fX)))

(let ([fac (mk-rec
           (lambda (fac)
             ; Exactly like original fac:
             (lambda (n)
               (if (zero? n)
                   1
                   (* n (fac (- n 1))))))]])
  (fac 10))
```



# Fibonacci

```
(let ([fib
      (mk-rec
       (lambda (fib)
         ; Usual fib:
         (lambda (n)
           (if (or (= n 0) (= n 1))
               1
               (+ (fib (- n 1))
                  (fib (- n 2)))))))]))
(fib 5))
```

# Sum

```
(let ([sum
      (mk-rec
       (lambda (sum)
         ; Usual sum:
         (lambda (l)
           (if (empty? l)
               0
               (+ (first l)
                  (sum (rest l)))))))]
      (sum '(1 2 3 4))))
```

# Implementing Recursion

```
{letrec {[fac {lambda {n}
            {if0 n
              1
              {* n
                {fac {- n 1}}}}}}]}
{fac 10}}
```

could be parsed the same as

```
{let {[fac
      {mk-rec
       {lambda {fac}
        {lambda {n}
         {if0 n
           1
           {* n
             {fac {- n 1}}}}}}}}]}
{fac 10}}
```

# Implementing Recursion

```
{letrec {[name rhs]}  
  body}
```

could be parsed the same as

```
{let {[name {mk-rec {lambda {name} rhs}}]}  
  body}
```

which is really

```
{{lambda {name} body}  
  {mk-rec {lambda {name} rhs}}}
```

which, writing out *mk-rec*, is really

```
{{lambda {name} body}  
  {{lambda {body-proc}  
    {let {[fX {fun {fX}  
      {let {[f {lambda {x}  
        {{fX fX} x}}]}  
      {body-proc f}}]}  
      {fX fX}}}  
    {lambda {name} rhs}}}
```