Part 1

Encodings

Using the minimal λ-calculus language we get

✓ functions

✓ local binding

✓ booleans

✓ numbers

Encodings

Using the minimal λ-calculus language we get

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

... and recursive functions?

Factorial in Shplait

```
block:
  def fac:
    fun (n):
      if n == 0
      | 1
      | n * fac(n - 1)
  fac(10)
```

Factorial in Shplait

```
block:
  def fac:
    fun (n):
      if n == 0
      | 1
      | n * fac(n - 1)
  fac(10)
```

`def` binds both its own right-hand side and expressions afterward

Factorial in Shplait

```
letrec fac = (fun (n):
                if n == 0
                | 1
                | n * fac(n - 1)):
  fac(10)
```

Factorial in Shplait

```
letrec fac = (fun (n):
                if n == 0
                | 1
                | n * fac(n - 1)):
  fac(10)
```

`letrec` has the shape of **`let`**,
but it has the binding structure of **`block`** plus **`def`**

Factorial in Shplait

```
let fac = (fun (n):
             if n == 0
             | 1
             | n * fac(n - 1)):
  fac(10)
```

Factorial in Shplait

```
let fac = (fun (n):
             if n == 0
             | 1
             | n * fac(n - 1)):
  fac(10)
```

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

Factorial

Overall goal: Implement `letrec` as syntactic sugar for Moe

```
letrec name = rhs:
  name
```

Factorial

Overall goal: Implement `letrec` as syntactic sugar for Moe

```
letrec name = rhs:
  name
```

Step 1: Encode `fac` in Shplait without `letrec`

Factorial

Overall goal: Implement `letrec` as syntactic sugar for Moe

```
letrec name = rhs:
  name
```

Step 1: Encode `fac` in Shplait without `letrec`

Step 2: Extract the *`rhs`* from within the encoding

```
.... fun (n):
       if n == 0
       | 1
       | n * fac(n - 1)
....
```

Factorial

Overall goal: Implement `letrec` as syntactic sugar for Moe

```
letrec name = rhs:
  name
```

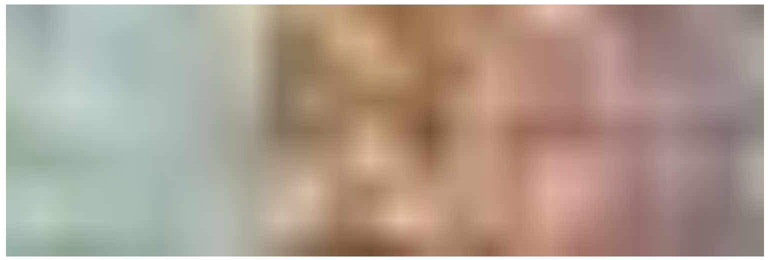
Step 1: Encode `fac` in Shplait without `letrec`

Step 2: Extract the *`rhs`* from within the encoding

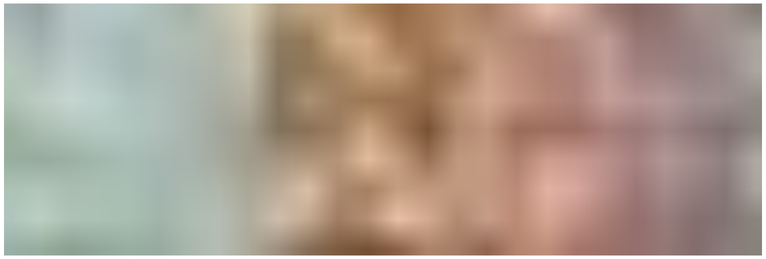
```
.... fun (n):
       if n == 0
       | 1
       | n * fac(n - 1)
....
```

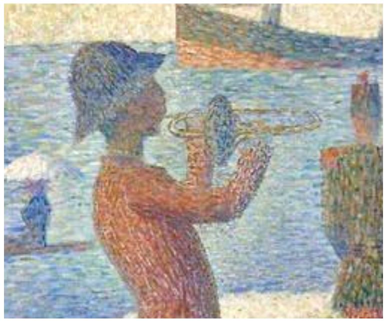
Step 3: Implement `letrec` as a `parse` transformation for Moe

This is Difficult...



This is Difficult...





```
mk_rec(f) = f(mk_rec(f))
```

Part 2

Factorial

Overall goal: Implement `letrec` as syntactic sugar for Moe

```
letrec name = rhs:
  name
```

Step 1: Encode `fac` in Shplait without `letrec`

Step 2: Extract the *`rhs`* from within the encoding

```
.... fun (n):
       if n == 0
       | 1
       | n * fac(n - 1)
....
```

Step 3: Implement `letrec` as a `parse` transformation for Moe

Factorial

```
let fac = (fun (n):
             if n == 0
             | 1
             | n * fac(n - 1)):
  fac(10)
```

Factorial

```
let fac = (fun (n):
              if n == 0
              | 1
              | n * fac(n - 1)):
  fac(10)
```

At the point that we call **`fac`**, obviously we have a binding for **`fac`**...

Factorial

```
let fac = (fun (n):
             if n == 0
             | 1
             | n * fac(n - 1)):
  fac(10)
```

At the point that we call **fac**, obviously we have a binding for **fac**...

... so pass it as an argument!

Factorial

```
let facX = (fun (facX, n):
              if n == 0
              | 1
              | n * fac(n - 1)):
  facX(facX, 10)
```

Factorial

```
let facX = (fun (facX, n):
              if n == 0
              | 1
              | n * facX(facX, n - 1)):
  facX(facX, 10)
```

Factorial

```
let facX = (fun (facX, n):
              if n == 0
              | 1
              | n * facX(facX, n - 1)):
  facX(facX, 10)
```

Wrap this to get **fac** back...

Factorial

```
let fac = (fun (n):
             let facX = (fun (facX, n):
                           if n == 0
                           | 1
                           | n * facX(facX, n - 1)):
               facX(facX, n)):
  fac(10)
```

Part 3

Factorial

Overall goal: Implement `letrec` as syntactic sugar for Moe

```
letrec name = rhs:
  name
```

Step 1: Encode `fac` in Shplait without `letrec`

Step 2: Extract the *`rhs`* from within the encoding

```
.... fun (n):
       if n == 0
       | 1
       | n * fac(n - 1)
....
```

Step 3: Implement `letrec` as a `parse` transformation for Moe

Factorial

```
let fac = (fun (n):
                let facX = (fun (facX, n):
                              if n == 0
                              | 1
                              | n * facX(facX, n - 1)):
                  facX(facX,  n)):
  fac(10)
```

Factorial

```
let fac = (fun (n):
             let facX = (fun (facX, n):
                           if n == 0
                           | 1
                           | n * facX(facX, n - 1)):
               facX(facX,  n)):
  fac(10)
```

But Moe has only single-argument functions...

Factorial

```
let fac = (fun (n):
             let facX = (fun (facX):
                           fun (n):
                             if n == 0
                             | 1
                             | n * facX(facX)(n - 1)):
               facX(facX)(n)):
  fac(10)
```

Factorial

```
let fac = (fun (n):
             let facX = (fun (facX):
                           fun (n):
                             if n == 0
                             | 1
                             | n * facX(facX)(n - 1)):
               facX(facX)(n)):
  fac(10)
```

Simplify: `fun (n): let f = ...: f(f)(n)`

⇒ `let f = ...: f(f)`...

Factorial

```
let fac = (let facX = (fun (facX):
                         // Almost looks like original fac:
                         fun (n):
                           if n == 0
                           | 1
                           | n * facX(facX)(n - 1)):
             facX(facX)):
  fac(10)
```

Factorial

```
let fac = (let facX = (fun (facX):
                        // Almost looks like original fac:
                        fun (n):
                          if n == 0
                          | 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 = (fun (facX):
                        let fac = facX(facX):
                          // Exactly like original fac:
                          fun (n):
                            if n == 0
                            | 1
                            | n * fac(n - 1)):
             facX(facX)):
  fac(10)
```

Factorial

```
let fac = (let facX = (fun (facX):
                         let fac = facX(facX):
                           // Exactly like original fac:
                           fun (n):
                             if n == 0
                             | 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

Factorial

```
let fac = (let facX = (fun (facX):
                         let fac = facX(facX):
                           // Exactly like original fac:
                           fun (n):
                             if n == 0
                             | 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 = (fun (facX):
                         let fac = (fun (x):
                                      facX(facX)(x)):
                           // Exactly like original fac:
                           fun (n):
                             if n == 0
                             | 1
                             | n * fac(n - 1)):
             facX(facX)):
  fac(10)
```

Factorial

```
let fac = (let facX = (fun (facX):
                         let fac = (fun (x):
                                      facX(facX)(x)):
                           // Exactly like original fac:
                           fun (n):
                             if n == 0
                             | 1
                             | n * fac(n - 1)):
             facX(facX)):
  fac(10)
```

Factorial

```
let fac = (let facX = (fun (facX):
                         let fac = (fun (x):
                                      facX(facX)(x)):
                           (fun (fac):
                              // Exactly like original fac:
                              fun (n):
                                if n == 0
                                | 1
                                | n * fac(n - 1))(fac)):
            facX(facX)):
  fac(10)
```

Factorial

```
let fac = let fX = (fun (fX):
                     let f = (fun (x):
                                fX(fX)(x)):
                       (fun (fac):
                          // Exactly like original fac:
                          fun (n):
                            if n == 0
                            | 1
                            | n * fac(n - 1))(f)):
            fX(fX):
  fac(10)
```

Factorial

```
let fac = let fX = (fun (fX):
                     let f = (fun (x):
                                fX(fX)(x)):
                      (fun (fac):
                         // Exactly like original fac:
                         fun (n):
                           if n == 0
                           | 1
                           | n * fac(n - 1))(f)):
            fX(fX)     :
  fac(10)
```

Factorial

```
def mk_rec:
  fun (body_proc):
```

```
let fac = let fX = (fun (fX):
                       let f = (fun (x):
          mk_rec(                      fX(fX)(x)):
                      (fun (fac):
                         // Exactly like original fac:
                         fun (n):
                           if n == 0  body_proc
                           | 1
                           | n * fac(n - 1)) )(f)):
                        :
             fX(fX)
  fac(10)
```

Factorial

```
def mk_rec:
  fun (body_proc):
```

```
        let fX = (fun (fX):
let fac =
                   let f = (fun (x):
         mk_rec((fun (fac):  fX(fX)(x)):
                  // Exactly like original fac:
                  fun (n):
                    if n == 0   body_proc
                    | 1
                    | n * fac(n - 1))(f)):
                                 )
          fX(fX)
  fac(10)
```

Factorial

```
def mk_rec:
  fun (body_proc):

    let fX = (fun (fX):
let fac =             let f = (fun (x):
          mk_rec((fun (fac):  fX(fX)(x)):
                 // Exactly like original fac:
                 fun (n): body_proc
                   if n == 0
                   | 1                 (f)):
      fX(fX)       | n * fac(n - 1))  )
                          :
  fac(10)
```

Factorial

```
def mk_rec:
  fun (body_proc):
```

```
        let fX = (fun (fX):
                    let f = (fun (x):
  let fac = mk_rec(fun (fac):  fX(fX)(x)):
                 // Exactly like original fac:
                 fun (n):body_proc
                   if n == 0
                                    (f)):
                   | 1
     fX(fX)
                   | n * fac(n - 1))
                             :           )

   fac(10)
```

Factorial

```
def mk_rec:
  fun (body_proc):

      let fX = (fun (fX):
                  let f = (fun (x):
                             fX(fX)(x)):
                  body_proc
                            (f)):
        fX(fX)

    let fac = mk_rec(
                 (fun (fac):
                   // Exactly like original fac:
                   fun (n):
                     if n == 0
                     | 1
                     | n * fac(n - 1))
                                         )
                            :
      fac(10)
```

Factorial

```
def mk_rec:
  fun (body_proc):

      let fX = (fun (fX):
                  let f = (fun (x):
                             fX(fX)(x)):
                  (fun (fac):
    let fac =mk_rec( // Exactly like original fac:
                     body_proc        (f)):
                     fun (n):
       fX(fX)          if n == 0
                       | 1
                       | n * fac(n - 1))
                                    :     )
      fac(10)
```

Factorial

```
def mk_rec:
  fun (body_proc):

      let fX = (fun (fX):
                  let f = (fun (x):
                             fX(fX)(x)):
                                    (fun (fac):
                                    body_proc (f)):
    let fac = mk_rec(  // Exactly like original fac:
      fX(fX)           fun (n):
                         if n == 0
                         | 1
                         | n * fac(n - 1))
                                         :  )

      fac(10)
```

Factorial

```
def mk_rec:
  fun (body_proc):
     let fX = (fun (fX):
                  let f = (fun (x):
                              fX(fX)(x)):
                   body_proc(f)):
      fX(fX)
  let fac = mk_rec((fun (fac):
                     // Exactly like original fac:
                     fun (n):
                       if n == 0
                       | 1
                       | n * fac(n - 1)) :)
    fac(10)
```

Factorial

```
def mk_rec:
  fun (body_proc):
    let fX = (fun (fX):
                let f = (fun (x):
                           fX(fX)(x)):
                  body_proc(f)):
      fX(fX)

let fac = mk_rec((fun (fac):
                    // Exactly like original fac:
                    fun (n):
                      if n == 0
                      | 1
                      | n * fac(n - 1))):
  fac(10)
```

Factorial

```
def mk_rec:
  fun (body_proc):
    let fX = (fun (fX):
                let f = (fun (x):
                           fX(fX)(x)):
                  body_proc(f)):
      fX(fX)

let fac = mk_rec((fun (fac):
                    // Exactly like original fac:
                    fun (n):
                      if n == 0
                      | 1
                      | n * fac(n - 1))):
  fac(10)
```

Factorial

```
let fac = mk_rec(fun (fac):
                   // Exactly like original fac:
                   fun (n):
                     if n == 0
                     | 1
                     | n * fac(n - 1)):
  fac(10)
```

Fibonnaci

```
let fib = mk_rec(fun (fib):
                   // Usual fib:
                   fun (n):
                     if n == 0 || n == 1
                     | 1
                     | fib(n - 1) + fib(n - 2)):
  fib(5)
```

Sum

```
let sum = mk_rec(fun (sum):
                   // Usual sum:
                   fun (lst):
                     match lst
                     | []: 0
                     | cons(f, rst): f + sum(rst)):
  sum([1, 2, 3, 4])
```

Part 4

Factorial

Overall goal: Implement `letrec` as syntactic sugar for Moe

```
letrec name = rhs:
  name
```

Step 1: Encode `fac` in Shplait without `letrec`

Step 2: Extract the *`rhs`* from within the encoding

```
.... fun (n):
       if n == 0
       | 1
       | n * fac(n - 1)
....
```

Step 3: Implement `letrec` as a `parse` transformation for Moe

Implementing Recursion

```
letrec fac = (fun (n):
                if n == 0
                | 1
                | n * fac(n - 1)):
  fac(10)
```

could be parsed the same as

```
let fac = mk_rec(fun (fac):
                   fun (n):
                     if n == 0
                     | 1
                     | n * fac(n - 1)):
  fac(10)
```

Implementing Recursion

```
letrec fac = (fun (n):
                if n == 0
                | 1
                | n * fac(n - 1)):
  fac(10)
```

could be parsed the same as

```
mk_rec = fun (body_proc):
           let fX = (fun (fX):
                      let f = (fun (x):
                                 fX(fX)(x)):
                        body_proc(f)):
             fX(fX)
```

```
let fac = mk_r
                        | n * fac(n - 1)):
  fac(10)
```

Implementing Recursion

```
letrec fac = (fun (n):
                if n == 0
                | 1
                | n * fac(n - 1)):
  fac(10)
```

could be parsed the same as

```
mk_rec = fun (body_proc):
           (fun (fx):
              fX(fX))(fun (fX):
                        (fun (f):
                           body_proc(f))(fun (x):
                                           fX(fX)(x)))
```

```
let fac = mk_r
                  | n * fac(n - 1)):
  fac(10)
```

Implementing Recursion

```
letrec name = rhs:
  body
```

could be parsed the same as

```
let name = mk_rec(fun (name): rhs):
  body
```

which is really

```
(fun (name): body)(mk_rec(fun (name): rhs))
```

which, writing out *mk_rec*, is really

```
(fun (name): body)((fun (body_proc):
                      let fX = (fun (fX):
                                  let f = (fun (x):
                                             fX(fX)(x)):
                                    body_proc(f)):
                        fX(fX))(fun (name): rhs))
```

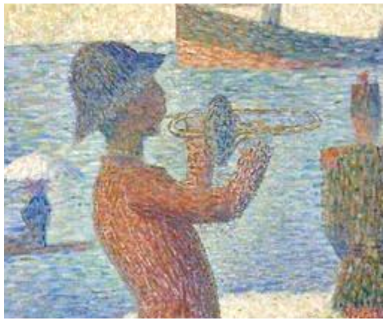
Part 5

The Big Picture

```
letrec name = rhs:
  body
```

⬇

```
(fun (name): body)((fun (body_proc):
                     let fX = (fun (fX):
                                 let f = (fun (x):
                                            fX(fX)(x)):
                                   body_proc(f)):
                       fX(fX))(fun (name): rhs))
```



Y Combinator

`mk_rec` is a ***fixed-point combinator***

```
fun (body_proc):
  (fun (fx):
     fX(fX))(fun (fX):
                (fun (f):
                   body_proc(f))(fun (x):
                                    fX(fX)(x)))
```

Y Combinator

`mk_rec` is a ***fixed-point combinator***

```
mk_rec(body_proc) = body_proc(mk_rec(body_proc))
```

Y Combinator

`mk_rec` is a ***fixed-point combinator***

```
mk_rec(body_proc) = body_proc(mk_rec(body_proc))
```

another is the ***Y combinator***

$$\mathtt{Y} \overset{\text{def}}{=} \mathtt{\lambda f{:}\ (\lambda(x){:}\ f(x\ x))(\lambda(x){:}\ f(x\ x))}$$

```
Y(f) = f(Y(f))
```

Y Combinator

`mk_rec` is a ***fixed-point combinator***

```
mk_rec(body_proc) = body_proc(mk_rec(body_proc))
```

another is the ***Y combinator***

$$\mathtt{Y} \overset{\text{def}}{=} \lambda\mathtt{f}\text{: } (\lambda(\mathtt{x})\text{: } \mathtt{f}(\mathtt{x}\ \mathtt{x}))(\lambda(\mathtt{x})\text{: } \mathtt{f}(\mathtt{x}\ \mathtt{x}))$$

```
Y(f) = f(Y(f))
```

See also *The Why of Y* (Gabriel) or *The Little Schemer* (Friedman & Felleisen)

Part 6

Example with Syntax Escapes

```
fun parse(s :: Syntax) :: Exp:
  match s
  | ....
  | 'let $id = $rhs: $body':
      parse('fun ($id): $body($rhs)')
  | ....
```