Part I
Encodings

Using the minimal \(\lambda\)-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?
def fac:
    fun (n):
        if n == 0
            | 1
            | n * fac(n - 1)
    fac(10)
Factorial in Shplait

```python
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

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

Factorial in Shplait

```shplait
letrec fac = (fun (n):
    if n == 0
    | 1
    | 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

```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 \texttt{letrec} as syntactic sugar for Moe

\begin{verbatim}
letrec name = rhs:
    name
\end{verbatim}
Factorial

Overall goal: Implement \texttt{letrec} as syntactic sugar for Moe

\begin{verbatim}
letrec name = rhs:
  name
\end{verbatim}

\textbf{Step 1:} Encode \texttt{fac} in Shplait without \texttt{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
     | 1
     | n * fac(n - 1)
....
```
Factorial

Overall goal: Implement \texttt{letrec} as syntactic sugar for \texttt{Moe}

\begin{verbatim}
letrec name = rhs:
  name
\end{verbatim}

\textbf{Step 1:} Encode \texttt{fac} in Shplait without \texttt{letrec}

\textbf{Step 2:} Extract the \texttt{rhs} from within the encoding

\begin{verbatim}
... fun (n):
  if n == 0
  | 1
  | n * fac(n - 1)
...
\end{verbatim}

\textbf{Step 3:} Implement \texttt{letrec} as a \texttt{parse} transformation for \texttt{Moe}
This is Difficult...
This is Difficult...

\[ \text{mk}_\text{rec}(f) = f(\text{mk}_\text{rec}(f)) \]
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

```plaintext
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

```ml
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 \texttt{letrec} as syntactic sugar for Moe

\begin{verbatim}
letrec name = rhs: 
  name
\end{verbatim}

**Step 1:** Encode \texttt{fac} in Shplait without \texttt{letrec}

**Step 2:** Extract the \texttt{rhs} from within the encoding

\begin{verbatim}
..... fun (n):
  if n == 0 |
   1 |
   n * fac(n - 1)
.....
\end{verbatim}

**Step 3:** Implement \texttt{letrec} as a \texttt{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

```plaintext
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

```javascript
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

```ml
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

```javascript
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

\[
\text{let } \text{fac} = (\text{let } \text{facX} = (\text{fun } (\text{facX}):
\hspace{1cm} \text{let } \text{fac} = (\text{fun } (x):
\hspace{1cm} \text{facX}((\text{facX})(x))):
\hspace{1cm} \text{// Exactly like original fac:}
\hspace{1cm} \text{fun } (n):
\hspace{1.5cm} \text{if } n == 0
\hspace{2cm} | 1
\hspace{2cm} | n * \text{fac}(n - 1)):
\hspace{1cm} \text{facX(facX))}):
\text{fac(10)}
\]
Factorial

```plaintext
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

```ml
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)
```

38
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

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

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

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

fac(10)
```
Factorial

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

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

fac(10)
```
def mk_rec:
    fun (body_proc):

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

        fac(10)
Factorial

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

        fac(10)
```

```
def mk_rec:
    fun (body_proc):

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

        let fac = mk_rec(fX(fX))
            // 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{
            // 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)
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

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

    fac(10)
```

Fibonacci

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

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

\[
\text{letrec } \text{name} = \text{rhs}:
\]

\[
\text{name}
\]

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

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

\[
\begin{align*}
\text{fun } (n): \\
& \text{if } n == 0 \\
& \mid 1 \\
& \mid n * \text{fac}(n - 1)
\end{align*}
\]

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

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

could be parsed the same as

```ocaml
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

let fac = mk_rec (fun (body_proc):
    let fX = (fun (fX):
        let f = (fun (x):
            fX(fX)(x)):
            body_proc(f)):
        fX(fX)
    | n * fac(n - 1)):
fac(10)
Implementing Recursion

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

could be parsed the same as

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

```
mk_rec = fun (body_proc):
    (fun (fx):
        fx(fX))(fun (fX):
            (fun (f):
                body_proc(f))(fun (x):
                    fx(fX)(x)))(fun (fx):
            n * fac(n - 1)):
```
Implementing Recursion

```plaintext
define (function name : rhs) (name : body)
define (mk_rec (function (name : rhs)) : body)
define (fun (name : body) (mk_rec (fun (name : rhs)))
```

which, writing out `mk_rec`, is really

```plaintext
(func (name : body) (fun (body_proc)) : body)
define (let fx = (fun (fx)) : body)
define (let f = (fun (x)) : body)
define (fx(fx)(x)) : body_proc(f)) : f(x)(fx)(fun (name : rhs))
```
Part 5
The Big Picture

\[
\text{letrec name = rhs:} \\
\text{body}
\]

\[
\text{(fun (name): body)((fun (body_proc):} \\
\text{ let fX = (fun (fX):} \\
\text{ let f = (fun (x):} \\
\text{ let fX(fX)(x)):} \\
\text{ body_proc(f)):} \\
\text{ fX(fX))(fun (name): rhs))}
\]
Y Combinator

\texttt{mk\_rec} is a \textit{fixed-point combinator}

\begin{verbatim}
fun (body_proc):
  (fun (fx):
    fx(fx))(fun (fx):
      (fun (f):
        body_proc(f))(fun (x):
          fx(fx)(x)))
\end{verbatim}
Y Combinator

`mk_rec` is a *fixed-point combinator*

\[
\text{mk}_\text{rec}(\text{body}_\text{proc}) = \text{body}_\text{proc}(\text{mk}_\text{rec}(\text{body}_\text{proc}))
\]
Y Combinator

\texttt{mk\_rec} is a \emph{fixed-point combinator}

\[
\texttt{mk\_rec(body\_proc) = body\_proc(mk\_rec(body\_proc))}
\]

another is the \textit{Y combinator}

\[
Y \overset{\text{def}}{=} \lambda f: (\lambda x: f(x\ x)) (\lambda x: f(x\ x))
\]

\[
Y(f) = f(Y(f))
\]
**Y Combinator**

\texttt{mk\_rec} is a \textit{fixed-point combinator}

\[
\texttt{mk\_rec(body\_proc)} = \texttt{body\_proc(mk\_rec(body\_proc))}
\]

another is the \textit{Y combinator}

\[
Y \overset{\text{def}}{=} \lambda f: (\lambda x: f(x\ x)) (\lambda x: f(x\ x))
\]

\[
Y(f) = f(Y(f))
\]

See also \textit{The Why of Y} (Gabriel) or \textit{The Little Schemer} (Friedman & Felleisen)
Example with Syntax Escapes

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