Part I

Encodings

Using the minimal λ-calculus language we get

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

... and recursive functions?

Factorial in Plait

```
(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 in Plait

```
(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 in Plait

```
(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

Factorial

Overall goal: Implement **`letrec`** as syntactic sugar for Curly

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

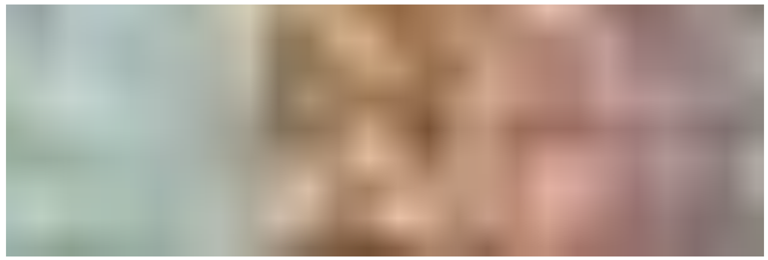
Step 1: Implement `fac` in Plait without **`letrec`**

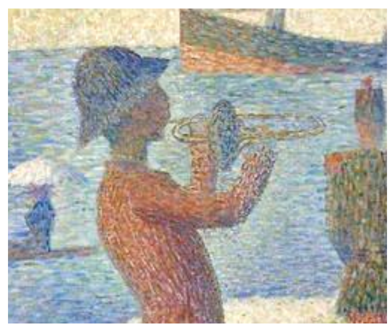
Step 2: Isolate the ***`rhs`***

```
.... {lambda {n}
       {if {zero? n}
           1
           {* n {fac {- n 1}}}}} ....
```

Step 3: Surrounding as a `parse` transformation for Curly

This is Difficult...





Part 2

Factorial

Overall goal: Implement **`letrec`** as syntactic sugar for Curly

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

Step 1: Implement `fac` in Plait without **`letrec`**

Step 2: Isolate the *`rhs`*

```
.... {lambda {n}
       {if {zero? n}
           1
           {* n {fac {- n 1}}}}} ....
```

Step 3: Surrounding as a `parse` transformation for Curly

Factorial

```
(let ([fac
        (lambda (n)
          (if (zero? n)
              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
       (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))
```

Part 3

Factorial

Overall goal: Implement **`letrec`** as syntactic sugar for Curly

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

Step 1: Implement `fac` in Plait without **`letrec`**

Step 2: Isolate the *`rhs`*

```
.... {lambda {n}
       {if {zero? n}
           1
           {* n {fac {- n 1}}}}} ....
```

Step 3: Surrounding as a `parse` transformation for Curly

Factorial

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

But Curly has only single-argument functions...

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)))`

⇒ `(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))
```

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))
```

Factorial

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

Factorial

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

Factorial

```
(define mk-rec
  (lambda (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))
```

Factorial

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

Fibonnaci

```
(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
                (+ (fst l)
                   (sum (rest l)))))))])
  (sum `(1 2 3 4)))
```

Part 4

Factorial

Overall goal: Implement **`letrec`** as syntactic sugar for Curly

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

Step 1: Implement `fac` in Plait without **`letrec`**

Step 2: Isolate the ***`rhs`***

```
.... {lambda {n}
       {if {zero? n}
           1
           {* n {fac {- n 1}}}}} ....
```

Step 3: Surrounding as a `parse` transformation for Curly

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 {[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}}
```

```
mk-rec = {lambda {body-proc}
           {let {[fX
                  {lambda {fX}
                    {let {[f {lambda {x}
                               {{fX fX} x}}]}
                      {body-proc f}}}]}
             {fX fX}}}
```

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}}
```

```
mk-rec = {lambda {body-proc}
           {{lambda {fx} {fX fX}}
            {lambda {fX}
              {{lambda {f} {body-proc f}}
               {lambda {x}
                 {{fX fX} x}}}}}}
```

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}}}
```

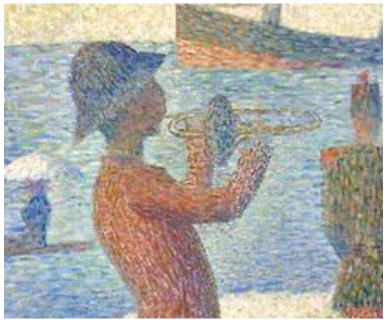
Part 5

The Big Picture

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

⬇

```
{{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}}}
```



Y Combinator

`mk-rec` is better known as the ***Y combinator***

```
{lambda {body-proc}
  {{lambda {fx} {fX fX}}
   {lambda {fX}
     {{lambda {f} {body-proc f}}
      {lambda {x}
        {{fX fX} x}}}}}}
```

Y Combinator

`mk-rec` is better known as the ***Y combinator***

```
Y ≝ (λ (g)
     {(λ (fx) {fX fX})
      (λ (fX)
        {(λ (f) {g f})
         (λ (x)
           {{fX fX} x})})})
```

a.k.a. the ***fixpoint operator***

```
{Y (lambda (f_in) f_out)}
```

Y Combinator

`mk-rec` is better known as the ***Y combinator***

```
Y ≝ (λ (g)
      {(λ (fx) {fX fX})
       (λ (fX)
         {(λ (f) {g f})
          (λ (x)
            {{fX fX} x})})})
```

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

Part 6

Example with Quasiquote Escapes

```
(define (parse [s : S-Exp]) : Exp
  ....
  [(s-exp-match? `{let {[SYMBOL ANY]} ANY} s)
   (let ([bs (s-exp->list (first
                            (s-exp->list (second
                                           (s-exp->list s)))))])
     (let ([name (first bs)]
           [rhs (second bs)]
           [body (third (s-exp->list s))])
       (parse `{{lambda {,name} ,body}
                ,rhs})))]
  ....)
```