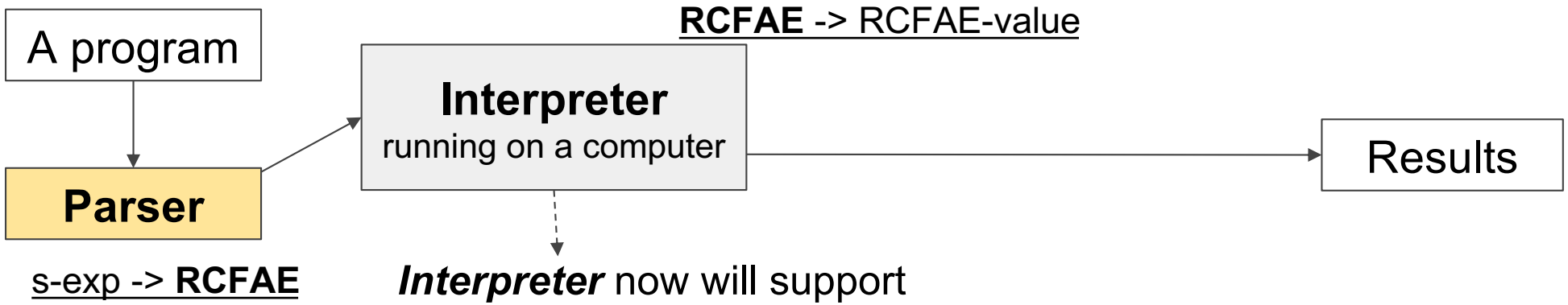
ITP30011
Implementing Recursion

Lecture16
JC

Big Picture



Interpreter now will support

(1) Substitution
(2) Function
(3) Deferring Substitution
(4) First-class Functions
(5) Laziness
(6) Recursion and its Implementation

RCFAE: Concrete Syntax

<RCFAE> ::= <num>
| {+ <RCFAE> <RCFAE>}
| {- <RCFAE> <RCFAE>}
| {* <RCFAE> <RCFAE>}
| <id>
| {fun {<id>} <RCFAE>}
| {<RCFAE> <RCFAE>}
| {if0 <RCFAE> <RCFAE> RCFAE>}
~~| {rec {<id> <RCFAE>} <RCFAE>}~~

<RCFAE> ::= <num>
| {+ <RCFAE> <RCFAE>}
| {- <RCFAE> <RCFAE>}
| {* <RCFAE> <RCFAE>}
| <id>
| {fun {<id} <RCFAE>}
| {<RCFAE> <RCFAE>}
| {if0 <RCFAE> <RCFAE> RCFAE>}
| {rec {<id> <RCFAE>} <RCFAE>}

Using the existing syntax vs. Adding new syntax 'rec'

RCFAE: Concrete Syntax

Using the existing syntax vs.

```
{with {fac {with {facX {fun {facY}
                         {with {fac {fun {x}
                                     {{facY facY} x}}}
                               {fun {n}
                                    {if0 n
                                         1
                                         {* n {fac {- n 1}}}}}}}
                               {facX facX}}}
      {fac 10}}
```

```
{with {fac {fun {n}
                {if0 n
                     1
                     {* n {fac {- n 1}}}}}}
      {fac 10}}
```

RCFAE: Concrete Syntax

Using the existing syntax vs. Adding new syntax 'rec'

```
{with {fac {with {facX {fun {facY}
                         {with {fac {fun {x}
                                      {{facY facY} x}}}
                               {fun {n}
                                    {if0 n
                                         1
                                         {* n {fac {- n 1}}}}}}}
                               {facX facX}}}
      {fac 10}}
```

```
{rec {fac {fun {n}
               {if0 n
                    1
                    {* n {fac {- n 1}}}}}}
     {fac 10}}
```

RCFAE: Concrete Syntax

Using the existing syntax vs. Adding new syntax 'rec'

```
{with {fac {with {facX {fun {facY}
                         {with {fac {fun {x}
                                     {{facY facY} x}}}
                               {fun {n}
                                    {if0 n
                                         1
                                         {* n {fac {- n 1}}}}}}}
                               {facX facX}}}
      {fac 10}}
```

```
{rec {fac {fun {n}
               {if0 n
                    1
                    {* n {fac {- n
1}}}}}}
     {fac 10}}
```

Do not need to significantly update our interpreter.

vs. Need to update our interpreter to support this syntax.

RCFAE: Concrete Syntax

Using the existing syntax vs. Adding new syntax 'rec'

```
{with {fac {with {facX {fun {facY}
                         {with {fac {fun {x}
                                      {{facY facY} x}}}
                               {fun {n}
                                    {if0 n
                                         1
                                         {* n {fac {- n 1}}}}}}}
                               {facX facX}}}
      {fac 10}}
```

```
{rec {fac {fun {n}
               {if0 n
                    1
                    {* n {fac {- n
1}}}}}}
     {fac 10}}
```

Do not need to significantly update our interpreter.

vs. Need to update our interpreter to support this syntax.

Code in concrete syntax is complicated.

vs. Code is intuitive and simpler.

What is your choice for developers??

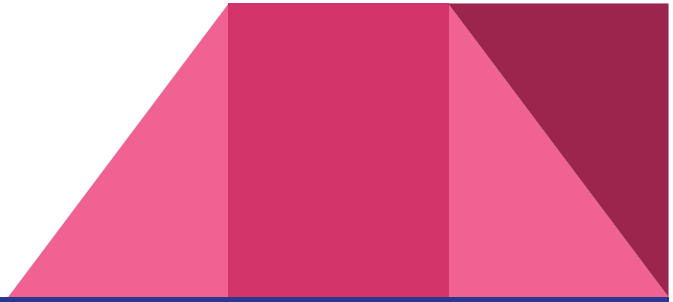
Leave an interpreter as it is. vs. Simple code

RCFAE: Concrete Syntax

<RCFAE> ::= <num>
| {+ <RCFAE> <RCFAE>}
| {- <RCFAE> <RCFAE>}
| <id>
| {fun {<id>} <RCFAE>}
| {<RCFAE> <RCFAE>}
| {if0 <RCFAE> <RCFAE> <RCFAE>}
| {rec {<id> <RCFAE>} <RCFAE>}

Example

```
{rec {count {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}}
                                                {count 8}}
```



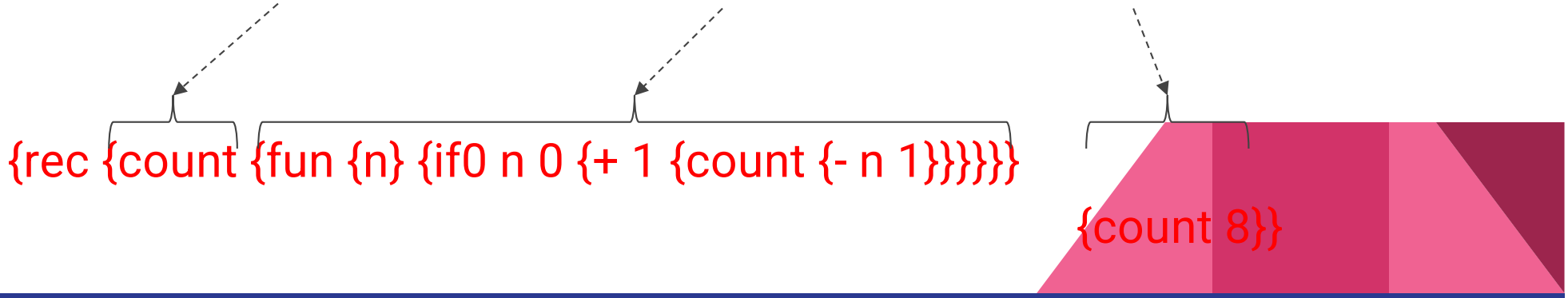
Example

{rec {count {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}}
{count 8}}

{count 8}
⇒ {+ 1 {count {- n 1}}} ⇒ {+ 1 {count 7}}
⇒ {+ 1 {+ 1 {count 6}}}
⇒ …
⇒ {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {count 0}}}}....}
⇒ {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 0}}}....}
⇒ …
⇒ 8

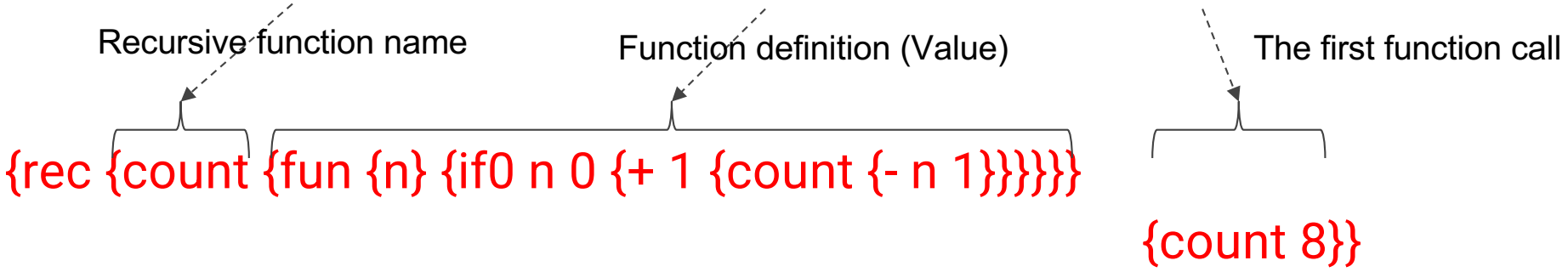
RCFAE: Abstract Syntax

```
(define-type RCFAE
   [num (n number?)]
   [add (lhs RCFAE?) (rhs RCFAE?)]
   [sub (lhs RCFAE?) (rhs RCFAE?)]
   [id (name symbol?)]
   [fun (param symbol?) (body RCFAE?)]
   [app (fun-expr RCFAE?) (arg-expr RCFAE?)]
   [if0 (test-expr RCFAE?)
        (then-expr RCFAE?) (else-expr RCFAE?)]
   [rec (name symbol?) (named-expr RCFAE?) (fst-call RCFAE?)])
```



RCFAE: Abstract Syntax

```
(define-type RCFAE
   [num (n number?)]
   [add (lhs RCFAE?) (rhs RCFAE?)]
   [sub (lhs RCFAE?) (rhs RCFAE?)]
   [id (name symbol?)]
   [fun (param symbol?) (body RCFAE?)]
   [app (fun-expr RCFAE?) (arg-expr RCFAE?)]
   [if0 (test-expr RCFAE?)
        (then-expr RCFAE?) (else-expr RCFAE?)]
   [rec (name symbol?) (named-expr RCFAE?) (fst-call RCFAE?)])
```

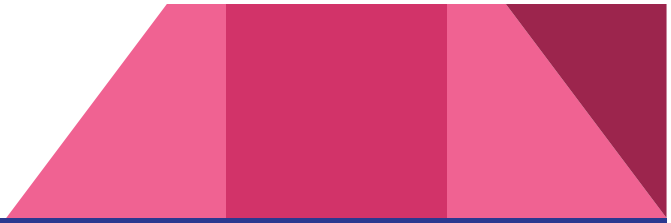


RCFAE: Interpreter

```
; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp rcfae ds)
   (type-case RCFAE rcfae
      [num (n) (numV n)]
      [add  (l r) (num+ (interp l ds) (interp r ds))]
      [sub  (l r) (num- (interp l ds) (interp r ds))]
      [id     (name) (lookup name ds)]
      [fun   (param body-expr) (closureV param body-expr ds)]
      [app  (f a) (local [(define ftn (interp f ds))]
                         (interp (closureV-body ftn)
                                 (aSub (closureV-param ftn)
                                       (interp a ds)
                                       (closureV-ds ftn)))))]
      [if0   (test-expr then-expr else-expr) ...]
      [rec  (bound-id named-expr fst-call) ...]))
```

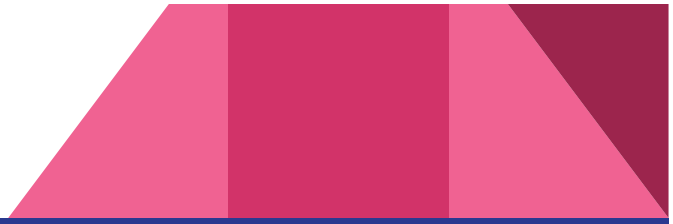
RCFAE: Interpreter

```
; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp rcfae ds)
   (type-case RCFAE rcfae
      …
      [if0   (test-expr then-expr else-expr)
             … (interp test-expr ds)
             … (interp then-expr ds)
             … (interp else-expr ds) …]
      [rec  (bound-id named-expr fst-call)
             …]))
```



RCFAE: Interpreter

```
; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp rcfae ds)
   (type-case RCFAE rcfae
      ...
      [if0   (test-expr then-expr else-expr)
             (if (numzero? (interp test-expr ds))
                 (interp then-expr ds)
                 (interp else-expr ds))]
      [rec  (bound-id named-expr fst-call)
            ...]))
```

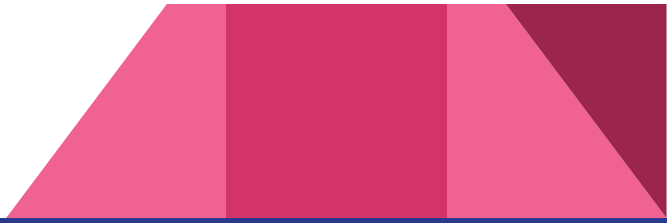


RCFAE: Interpreter

```
; numzero? :  RCFAE-Value -> boolean
(define (numzero? n)
   (zero? (numV-n n)))
```

RCFAE: Interpreter

```
; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp rcfae ds)
   (type-case RCFAE rcfae
      ...
      [rec  (bound-id named-expr fst-call)
            ...]))
```



RCFAE: Interpreter

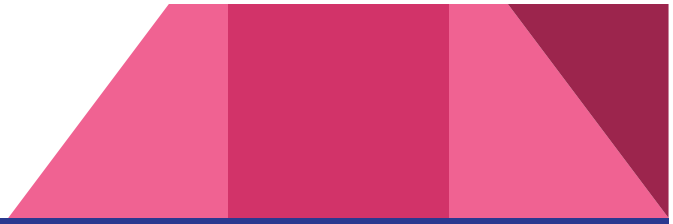
```
; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp rcfae ds)
   (type-case RCFAE rcfae
      ...
      [rec  (bound-id named-expr fst-call)
            (interp fst-call
                  (aSub bound-id
                        (interp named-expr ds)
                        ds))]
```

Can't interpret this because bound-id used in named-expr.

How can we solve this?

RCFAE: Interpreter

```
; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp rcfae ds)
   (type-case RCFAE rcfae
      ...
      [rec  (bound-id named-expr fst-call)
            ... (interp named-expr ds)
            ... (interp fst-call ds) ...]))
```



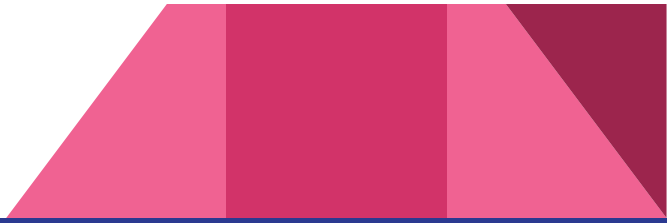
RCFAE: Interpreter

```
; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp rcfae ds)
   (type-case RCFAE rcfae
      …
      [rec  (bound-id named-expr fst-call)
            (local [(define new-ds (aRecSub bound-id
                                            …
                                            ds))]
              … (interp named-expr new-ds)
              … (interp fst-call new-ds) …]))
```

RCFAE: DefrdSub

```
(define-type DefrdSub
   [mtSub]
   [aSub       (name symbol?)
               (value RCFAE-Value?)
               (ds DefrdSub?)]
   [aRecSub  (name symbol?)
               (value-box (box/c RCFAE-Value?))
               (ds DefrdSub?)])

(define-type RCFAE-Value
   [numV       (n number?)]
   [closureV   (param Symbol?) (body RCFAE?) (ds DefrdSub?)])
```



RCFAE: Interpreter

```
; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp rcfae ds)
   (type-case RCFAE rcfae
      …
      [rec  (bound-id named-expr fst-call)
            (local [(define value-holder (box (numV 198)))
                    (define new-ds (aRecSub bound-id
                                            value-holder
                                            ds))]
              … (interp named-expr new-ds)
              … (interp fst-call new-ds) ...]))
```

* **Dummy value**: 198
(Just put an arbitrary number to initialize the value-holder.
If the program uses the identifier being bound before it has its real value, it'll get the dummy value as the result. But because we have assumed that the named expression is syntactically a function, this can't happen.

Example

{rec {count {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}}
(numV 198)
{count 8}}

{count 8}
⇒ {+ 1 {count {- n 1}}} ⇒ {+ 1 {count 7}}}
⇒ {+ 1 {+ 1 {count 6}}}
⇒ …
⇒ {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {count 0}}}}....}
⇒ {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 0}}}....}
⇒ …
⇒ 8

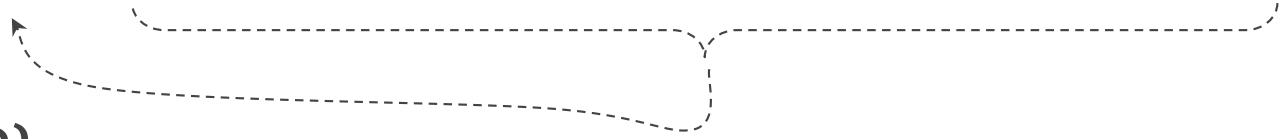
RCFAE: Interpreter

```
; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp rcfae ds)
   (type-case RCFAE rcfae
      ...
      [fun (param body-expr) (closureV param body-expr ds)]
      ...
      [rec  (bound-id named-expr fst-call)
            (local [(define value-holder (box (numV 198)))
                    (define new-ds (aRecSub bound-id
                                            value-holder   ds))]
                   (begin
                      (set-box! value-holder (interp named-expr new-ds))
                      (interp fst-call new-ds)))]))
```

```
{rec {count {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}}}
                                                  {count 8}}
```

Example

{rec {count {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}}
{count 8}}



{count 8}

⇒ {+ 1 {count {- n 1}}} ⇒ {+ 1 {count 7}}}

⇒ {+ 1 {+ 1 {count 6}}}

⇒ …

⇒ {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {count 0}}}}}}}}....}

⇒ {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 0}}}}}}}....}

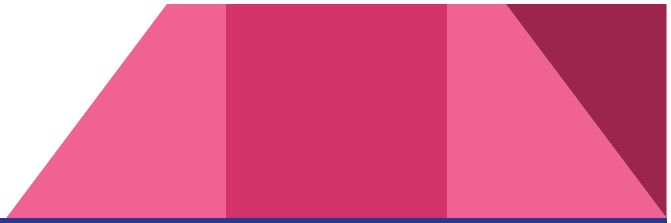
⇒ ...

⇒ 8

RCFAE: DefrdSub

```
(define-type DefrdSub
   [mtSub]
   [aSub       (name symbol?)
               (value RCFAE-Value?)
               (ds DefrdSub?)]
   [aRecSub  (name symbol?)
               (value-box (box/c RCFAE-Value?))
               (ds DefrdSub?)])

(define-type RCFAE-Value
   [numV       (n number?)]
   [closureV   (param Symbol?) (body RCFAE?) (ds DefrdSub?)])
```



RCFAE: Lookup

```
; lookup : symbol DefrdSub -> RCFAE-Value
(define (lookup name ds)
   (type-case DefrdSub ds
      [mtSub  () (error 'lookup "free variable")]
      [aSub    (sub-name val rest-ds)
                        (if (symbol=? sub-name name)
                            val
                            (lookup name rest-ds))]
      [aRecSub (sub-name val-box rest-ds)
                  (if (symbol=? sub-name name)
                      (unbox val-box)
                      (lookup name rest-ds))]))
```

Boxes in DrScheme

A box is like a single-element vector, normally used as minimal mutable storage.

http://docs.racket-lang.org/reference/boxes.html

- box: (define value-holder (box (numV 198)))
- set-box!
  (set-box! value-holder (interp named-expr new-ds))
- unbox: (unbox val-box)
- box/c: (value-box (box/c RCFAE-Value?))

Example Run!

```
(run '{rec {count {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}}
           {count 8}} (mtSub))
```

```
f           =
fun-expr    =
fst-call    =
value-holder =
new-ds      =
```

Example Run!

```
[rec (f fun-expr fst-call)
     (local [(define value-holder (box (numV 198)))
             (define new-ds (aRecSub f value-holder ds))]
       (begin
         (set-box! value-holder (interp fun-expr new-ds))
         (interp fst-call new-ds)))]
```

```
(run '{rec {count {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}}
           {count 8}} (mtSub))
```

f =
fun-expr =
fst-call =
value-holder =
new-ds =

Example Run!

```
[rec (f fun-expr fst-call)
     (local [(define value-holder (box (numV 198)))
             (define new-ds (aRecSub f value-holder ds))]
       (begin
         (set-box! value-holder (interp fun-expr new-ds))
         (interp fst-call new-ds)))]
```

```
(run '{rec {count {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}}
           {count 8}} (mtSub))
```

```
f            = count
fun-expr     = {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}
fst-call     = {count 8}
value-holder = [numV 198]
new-ds       = (aRecSub 'count value-holder (mtSub))
```

Example Run!

```
[rec (f fun-expr fst-call)
     (local [(define value-holder (box (numV 198)))
             (define new-ds (aRecSub f value-holder ds))]
       (begin
         (set-box! value-holder (interp fun-expr new-ds))
         (interp fst-call new-ds)))]
```

(interp fun-expr new-ds)

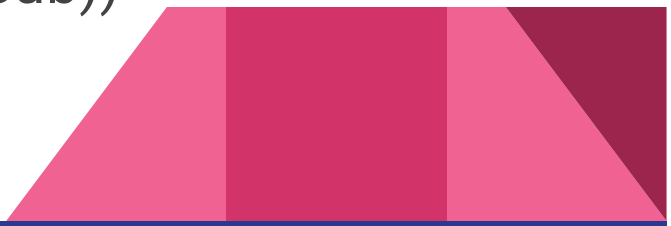
```
fun-expr       = {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}
fst-call       = {count 8}
value-holder = [numV 198]
new-ds         = (aRecSub 'count value-holder (mtSub))
```

Example Run!

[fun (param body-expr) (closureV param body-expr ds)]

(interp fun-expr new-ds)

fun-expr = {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}
fst-call = {count 8}
value-holder = [numV 198]
new-ds = (aRecSub 'count value-holder (mtSub))



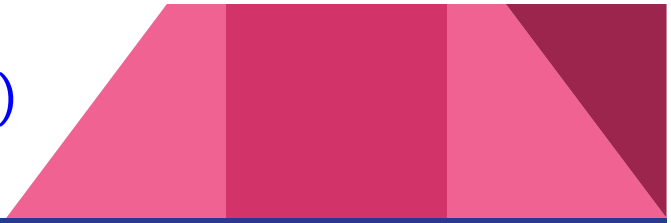
Example Run!

[fun (param body-expr) (closureV param body-expr ds)]

(interp fun-expr new-ds)

fun-expr = {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}
fst-call = {count 8}
value-holder = [numV 198]
new-ds = (aRecSub 'count value-holder (mtSub))

(interp fun-expr new-ds)
= (closureV 'n (if0 n 0 (+ 1 (count (- n 1)))) new-ds)



Example Run!

```
[rec (f fun-expr fst-call)
     (local [(define value-holder (box (numV 198)))
             (define new-ds (aRecSub f value-holder ds))]
       (begin
         (set-box! value-holder (interp fun-expr new-ds))
         (interp fst-call new-ds)))]
```

(interp funexpr new-ds)

fun-expr = {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}
fst-call = {count 8}
value-holder = [numV 198]
new-ds = (aRecSub 'count value-holder (mtSub))

(interp fun-expr new-ds)
= (closureV 'n (if0 n 0 (+ 1 (count (- n 1)))) new-ds)

Example Run!

```
[rec (f fun-expr fst-call)
     (local [(define value-holder (box (numV 198)))
             (define new-ds (aRecSub f value-holder ds))]
       (begin
         (set-box! value-holder (interp fun-expr new-ds))
         (interp fst-call new-ds)))]
```

```
(set-box! value-holder (interp fun-expr new-ds))
```

```
fun-expr     = {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}
fst-call     = {count 8}
value-holder = [numV 198]
new-ds       = (aRecSub 'count value-holder (mtSub))
```

```
(interp fun-expr new-ds)
    = (closureV 'n (if0 n 0 (+ 1 (count (- n 1)))) new-ds)
```

Example Run!

```
[rec (f fun-expr fst-call)
     (local [(define value-holder (box (numV 198)))
             (define new-ds (aRecSub f value-holder ds))]
       (begin
         (set-box! value-holder (interp fun-expr new-ds))
         (interp fst-call new-ds)))]
```

(set-box! value-holder (interp fun-expr new-ds))

fun-expr = {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}
fst-call = {count 8}
value-holder = [(closureV 'n '{if0 n 0 {+ 1 {count {- n 1}}}} new-ds)]
new-ds = (aRecSub 'count value-holder (mtSub))

(interp fun-expr new-ds)
= (closureV 'n (if0 n 0 (+ 1 (count (- n 1)))) new-ds)

Example Run!

```
[rec (f fun-expr fst-call)
     (local [(define value-holder (box (numV 198)))
             (define new-ds (aRecSub f value-holder ds))]
       (begin
         (set-box! value-holder (interp fun-expr new-ds))
         (interp fst-call new-ds)))]
```

(interp fst-call new-ds)

fun-expr = {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}
fst-call = {count 8}
value-holder = [(closureV 'n '{if0 n 0 {+ 1 {count {- n 1}}}} new-ds)]
new-ds = (aRecSub 'count value-holder (mtSub))

(interp fun-expr new-ds)
= (closureV 'n (if0 n 0 (+ 1 (count (- n 1)))) new-ds)

Example Run!

```
[app  (f a) (local [(define ftn (interp f ds))]
                   (interp (closureV-body ftn)
                           (aSub (closureV-param ftn)
                                 (interp a ds)
                                 (closureV-ds ftn))))]
```

(interp fst-call new-ds)

fun-expr = {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}
fst-call = {count 8}
value-holder = [(closureV 'n '{if0 n 0 {+ 1 {count {- n 1}}}} new-ds)]
new-ds = (aRecSub 'count value-holder (mtSub))

(interp fun-expr new-ds)
= (closureV 'n (if0 n 0 (+ 1 (count (- n 1)))) new-ds)

Example Run!

```
    [if0   (test-expr then-expr else-expr)
           (if (numzero? (interp test-expr ds))
               (interp then-expr ds)
               (interp else-expr ds))]

(interp (closureV-body ftn)
          (aSub (closureV-param ftn)
                (interp a ds)
                (closureV-ds ftn)))

fun-expr       = {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}
fst-call       = {count 8}
value-holder = [(closureV 'n '{if0 n 0 {+ 1 {count {- n 1}}}} new-ds)]
new-ds         = (aRecSub 'count value-holder (mtSub))
```

```
(interp fun-expr new-ds)
      = (closureV 'n (if0 n 0 (+ 1 (count (- n 1)))) new-ds)
```

Example

{rec {count {fun {n} {if0 n 0 {+ 1 {count {- n 1}}}}}}
{count 8}}

{count 8}
⇒ {+ 1 {count {- n 1}}} ⇒ {+ 1 {count 7}}}
⇒ {+ 1 {+ 1 {count 6}}}
⇒ ...
⇒ {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {count 0}}}}....}
⇒ {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 0}}}....}
⇒ {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 1}}}....}
⇒ {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 {+ 1 2}}....}
...
⇒ {+ 1 7}
⇒ 8

Topics we cover and schedule (tentative)

- Racket tutorials (L2,3, HW)
- Modeling languages (L4,5, HW
- Interpreting arithmetic (L5)
- Language principles
  - **Substitution** (L6, HW)
  - **Function** (L7)
  - **Deferring Substitution** (L8,L9)
  - **First-class Functions** (L10-12)
  - **Laziness** (L13, L14)
  - **Recursion** (L15, L16)
  - Mutable data structures
  - Variables
  - Continuations
- Guest Video Lecture

No class: October 2 (Fri, Chuseok), October 9 (Fri, Hangul day)
Online only class can be provided.

JC
jcnam@handong.edu
https://lifove.github.io

TODO

Read Chapter 13. Mutable Data Structures

* Slides are from Prof. Sukyoung Ryu's PL class in 2018 Spring or created by JC based on the main text book.