ITP30011 L10/L11/12
First-class Functions

Lecture10
JC

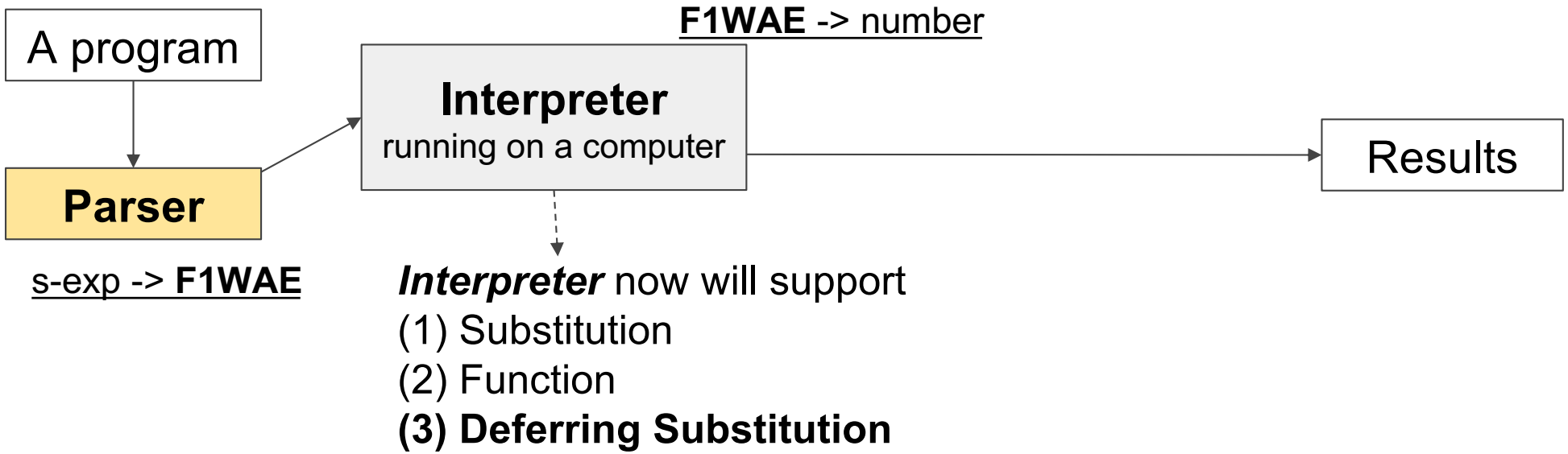
PLAI Ch 6 First class functions
http://cs.brown.edu/~sk/Publications/Books/ProgLangs/2007-04-26/plai-2007-04-26.pdf

PLAI 2nd Ed. Ch7 Functions anywhere!
http://cs.brown.edu/courses/cs173/2012/book/higher-order-functions.html

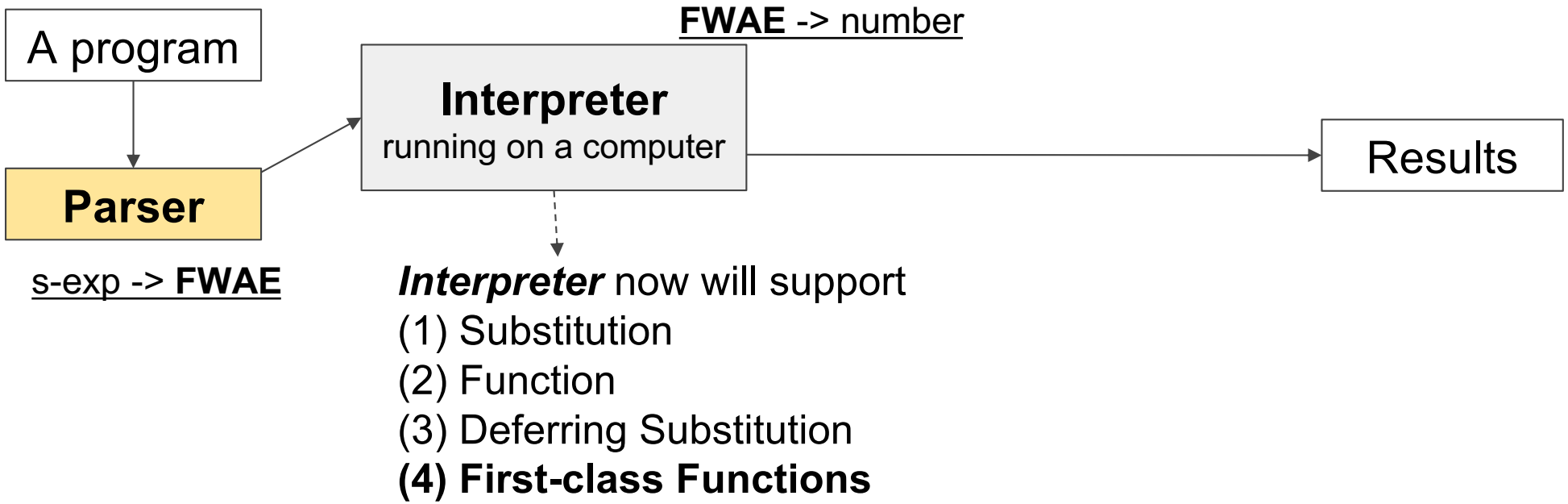
Q&A

Time complexity of the deferred substitution algorithm???

Big Picture (modeling languages: substitution)



Big Picture (modeling languages: substitution)



First-order Functions

"Functions are not values in languages."
: Names must be given for use in the remainder of a program.

⇒ F1WAE

Higher-order Functions

"Functions can return other functions as values."

First-class Functions

"Functions are values with all the rights of other values."
: Can be supplied as the value of arguments to functions.
: Can be returned by functions as answers
: Can stored in data structures.

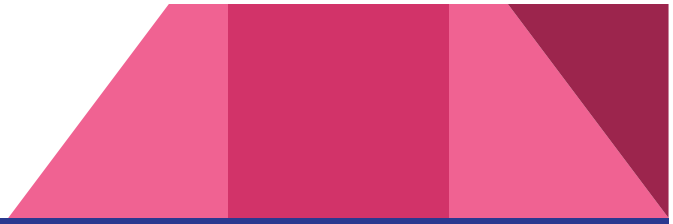
⇒ Full power of functions!!!!

Any real world examples in any PLs?

- https://en.wikipedia.org/wiki/First-class_function
- https://developer.mozilla.org/en-US/docs/Glossary/First-class_Function
- https://dzone.com/articles/java-lambda-expressions-functions-as-first-class-citizens
- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- https://www.baeldung.com/java-8-lambda-expressions-tips
- http://tutorials.jenkov.com/java/lambda-expressions.html
- https://stackoverflow.com/questions/5178068/what-is-a-first-class-citizen-function
- https://stackoverflow.com/questions/2755445/how-can-i-write-an-anonymous-function-in-java

How does First-class functions look like in our language we are going to make?

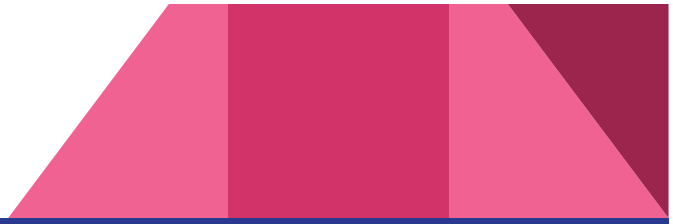
```
{with {double {fun {x} {+ x x}}
        {+ {double 10}
           {double 5}}}
```



How does First-class functions look like in our language we are going to make?

```
{with {double {fun {x} {+ x x}}
        {+ {double 10}
           {double 5}}}

; Equivalent program by F1WAE
{deffun {f x} {+ x x}}
{+ {f 10} {f 5}}
```



Recall F1WAE...

- Concrete syntax of F1WAE

  <FunDef> ::= {deffun {<id> <id>} <F1WAE>}

  <F1WAE> ::= <num>

  | {+ <F1WAE> <F1WAE>}

  | {- <F1WAE> <F1WAE>}

  | {with {<id> <F1WAE>} <F1WAE>}

  | <id>

  | {<id> <F1WAE>}

{deffun {twice x} {+ x x}}
{- 20 {twice 10}}
{- 20 {twice 17}}
{- 20 {twice 3}}

Let's start from WAE to add functions

- We name our language that supports first-class functions as FWAE

<FWAE> ::= <num>
| {+ <FWAE> <FWAE>}
| {- <FWAE> <FWAE>}
| {with {<id> <FWAE>} <FWAE>}
| <id>
| {<id> <FWAE>}
| {fun {<id>} <FWAE>}

Let's start from WAE to add functions

- We name our language that supports first-class functions as FWAE

<FWAE> ::= <num>
| {+ <FWAE> <FWAE>}
| {- <FWAE> <FWAE>}
| {with {<id> <FWAE>} <FWAE>}
| <id>
| {<id> <FWAE>} ???
| {fun {<id>} <FWAE>} e.g., {fun {x} {+ x x}}

f(x) = x + x ⇒ f(5)
f = λ(x)x+x ⇒ f(5)
(λ(x)x+x)(5)

* Lambda calculus: https://en.wikipedia.org/wiki/Lambda_calculus
"Lambda calculus (also written as λ-calculus) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution."

Let's start from WAE to add functions

- We name our language that supports first-class functions as FWAE

<FWAE> ::= <num>
| {+ <FWAE> <FWAE>}
| {- <FWAE> <FWAE>}
| {with {<id> <FWAE>} <FWAE>}
| <id>
| {<id> <FWAE>} ???
| {fun {<id>} <FWAE>} e.g., {fun {x} {+ x x}}

{- 20 {??? 10}}
{- 20 {??? 17}}
{- 20 {??? 3}}

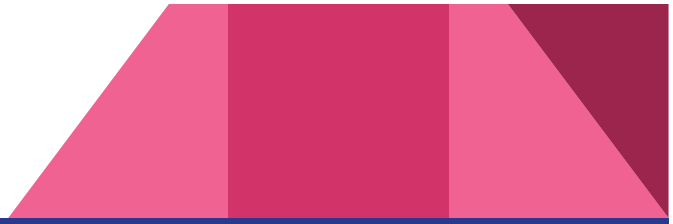
Let's start from WAE to add functions

- We name our language that supports first-class functions as FWAE

<FWAE> ::= <num>
| {+ <FWAE> <FWAE>}
| {- <FWAE> <FWAE>}
| {with {<id> <FWAE>} <FWAE>}
| <id>
| {<FWAE> <FWAE>}
| {fun {<id>} <FWAE>}

{with {f {fun {x} {+ x x}}}
{- 20 {f 10}}}

{- 20 {{fun {x} {+ x x}} 10}}



Example of FWAE Evaluation

10 ⇒ 10
{+ 1 2} ⇒ 3
{- 1 2} ⇒ -1
{with {x 7} {+ x 2}} ⇒ {+ 7 2} ⇒ 9
y ⇒ error: free identifier!
{fun {x} {+ 1 x}} ⇒ ???

Example of FWAE Evaluation

10 ⇒ 10

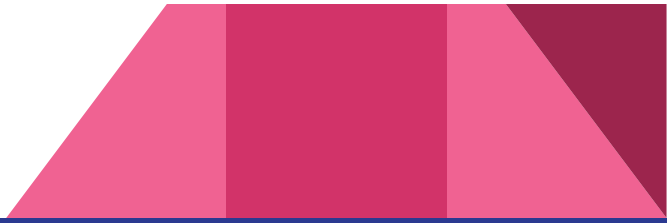
{+ 1 2} ⇒ 3

{- 1 2} ⇒ -1

{with {x 7} {+ x 2}} ⇒ {+ 7 2} ⇒ 9

y ⇒ error: free identifier!

{fun {x} {+ 1 x}} ⇒ {fun {x} {+ 1 x}}



Example of FWAE Evaluation

10 ⇒ 10
{+ 1 2} ⇒ 3
{- 1 2} ⇒ -1
{with {x 7} {+ x 2}} ⇒ {+ 7 2} ⇒ 9
y ⇒ error: free identifier!
{fun {x} {+ 1 x}} ⇒ {fun {x} {+ 1 x}}

Result is not always a number!
; interp FWAE … -> num

⇓

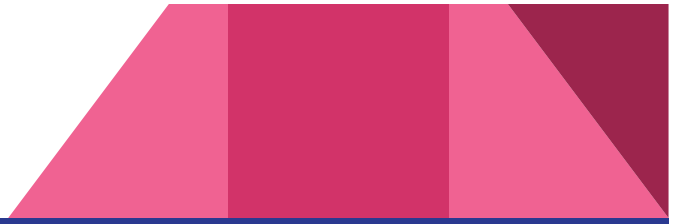
; interp FWAE … -> FWAE-Value

* These examples are just examples for explaining FWAE evaluation roughly for your intuitive understanding. You can't directly use these for actual test cases. (You need to modify them properly based on type definitions and your implementation.

Example FWAE Evaluation

```
    {with {f {fun {x} {+ 1 x}}} {f 3}}
⇒   {{fun {x} {+ 1 x}} 3}
⇒   {+ 1 3}
⇒   4
```

```
{1 2}                          ⇒ error: not a function!
{+ 1 {fun {x} 10}}             ⇒ error: not a number!
```



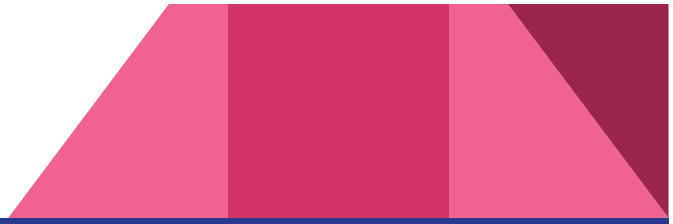
Let's start from WAE to add functions

- We name our language that supports first-class functions as FWAE

<FWAE> ::= <num>
| {+ <FWAE> <FWAE>}
| {- <FWAE> <FWAE>}
| {with {<id> <FWAE>} <FWAE>}
| <id>
| {<FWAE> <FWAE>}
| {fun {<id>} <FWAE>}

{with {f {fun {x} {+ x x}}}
{- 20 {f 10}}}

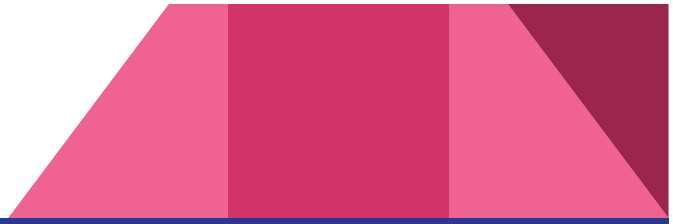
{- 20 {{fun {x} {+ x x}} 10}}



Example FWAE Evaluation

{with {f {fun {x} {+ 1 x}}} {f 3}}
⇒ {{fun {x} {+ 1 x}} 3}
⇒ {+ 1 3}
⇒ 4

{1 2} ⇒ error: not a function!
{+ 1 {fun {x} 10}} ⇒ error: not a number!

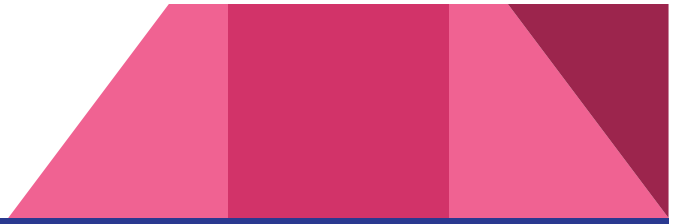


F1WAE: Abstract Syntax

```
(define-type FunDef
   [fundef    (fun-name symbol?) (arg-name symbol?) (body F1WAE?)])
(define-type F1WAE
   [num    (n number?)]
   [add    (lhs F1WAE?) (rhs F!WAE?)]
   [sub    (lhs F1WAE?) (rhs F!WAE?)]
   [with   (name symbol?) (named-expr F1WAE?) (body F1WAE?)]
   [id      (name symbol?)]
   [app   (ftn symbol?) (arg F1WAE?)])
```

```
(fundef   'identify 'x (id 'x))
(app 'identity (num 8))
```

```
(fundef 'twice 'x (add (id 'x) (id 'x)))
(app 'twice (num 10))
(app 'twice (num 17))
(app 'twice (num 3))
```



FWAE: Abstract Syntax

```
(define-type FWAE
   [num    (n number?)]
   [add    (lhs FWAE?) (rhs FWAE?)]
   [sub    (lhs FWAE?) (rhs FWAE?)]
   [with   (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [id     (name symbol?)]
   [fun    (param symbol?) (body FWAE?)]
   [app    (ftn FWAE?) (arg FWAE?)])
```

```
(fun 'x (add (id 'x) (id 'x)))
(app (id 'twice) (num 10))
(app (fun 'x (add (id 'x) (id 'x))) (num 10))
```

FWAE: Abstract Syntax

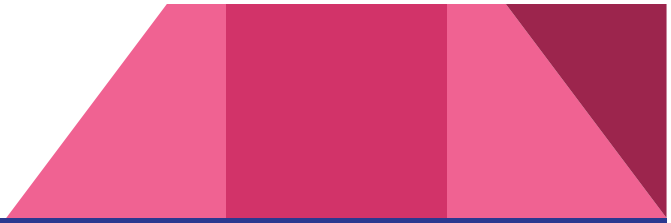
```
(define-type FWAE
   [num    (n number?)]
   [add    (lhs FWAE?) (rhs FWAE?)]
   [sub    (lhs FWAE?) (rhs FWAE?)]
   [with   (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [id     (name symbol?)]
   [fun    (param symbol?) (body FWAE?)]
   [app    (ftn FWAE?) (arg FWAE?)])

(test (parse '{fun {x} {+ x 1}})
              (fun 'x (add (id 'x) (num 1))))
```

FWAE: Abstract Syntax

```
(define-type FWAE
   [num    (n number?)]
   [add    (lhs FWAE?) (rhs FWAE?)]
   [sub    (lhs FWAE?) (rhs FWAE?)]
   [with   (name symbol?) (named-expr FWAE?) (body FWAE?)]
   [id     (name symbol?)]
   [fun    (param symbol?) (body FWAE?)]
   [app    (ftn FWAE?) (arg FWAE?)])

(test (parse '{{fun {x} {+ x 1}} 10})
              (app (fun 'x (add (id 'x) (num 1))) (num 10)))
```



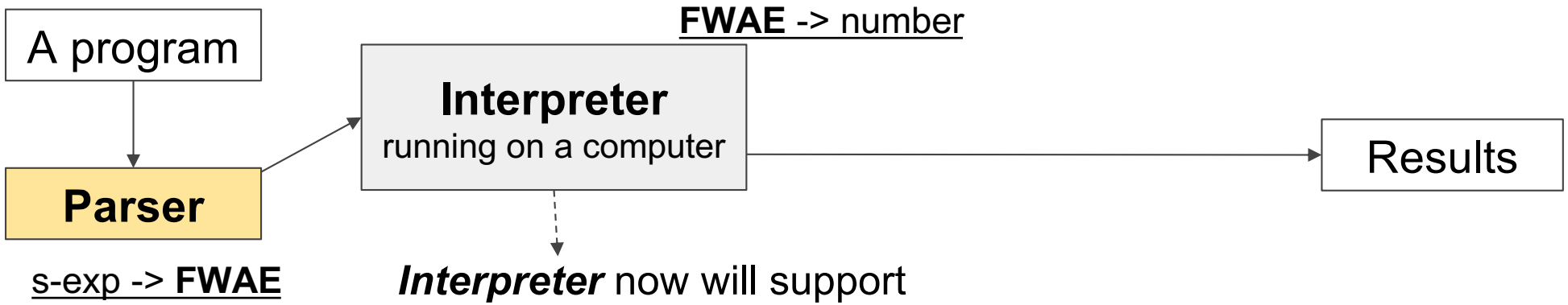
FWAE: Parser

```
; parse: sexp -> FWAE
; purpose: to convert sexp to FWAE
(define (parse sexp)
  (match sexp
    [(? number?)              (num sexp)]
    [(list '+ l r)            (add (parse l) (parse r))]
    [(list '- l r)            (sub (parse l) (parse r))]
    [(list 'with (list i v) e)  (with  i (parse v) (parse e))]
    [(? symbol?)              (id sexp)]
    [(list 'fun (list p) b)   (fun p (parse b))]  ;; e.g., {fun {x} {+ x 1}}
    [(list f a)               (app (parse f) (parse a))]
    [else                     (error 'parse "bad syntax: ~a" sexp)]))
```

ITP30011 L11
First-class Functions (2)

Lecture11
JC

Big Picture (modeling languages: substitution)



Interpreter now will support

(1) Substitution

(2) Function

(3) Deferring Substitution

(4) First-class Functions

F1WAE: Interpreter

```
; interp: F1WAE list-of-FuncDef -> number
(define (interp f1wae fundefs)
    (type-case F1WAE f1wae
        [num   (n)        n]
        [add   (l r)      (+ (interp l fundefs) (interp r fundefs))]
        [sub   (l r)      (- (interp l fundefs) (interp r fundefs))]
        [with  (i v e)    (interp (subst e i (interp v fundefs)) fundefs)]
        [id    (s)        (error 'interp "free identifier")]
        [app   (f a)      ...]))
```

FWAE: Interpreter

```
; interp: FWAE list-of-FuncDef -> number FWAE
(define (interp fwae)
    (type-case FWAE fwae
        [num   (n)        fwae]    ;{+ 3 4}  >> (num 7) rather than 7
        [add   (l r)       (num+ (interp l) (interp r))]
        [sub   (l r)       (num- (interp l) (interp r))]
        [with  (i v e)     (interp (subst e i (interp v)))]
        [id    (s)         (error 'interp "free identifier")]
        [fun   (p b)       ...]
        [app   (f a)       ...]))
```

FWAE: Interpreter

```
; interp: FWAE -> FWAE
(define (interp fwae)
    (type-case FWAE fwae
        [num   (n)       fwae]
        [add   (l r)      (num+ (interp l) (interp r))]
        [sub   (l r)      (num- (interp l) (interp r))]
        [with  (i v e)    (interp (subst e i (interp v)))]
        [id    (s)        (error 'interp "free identifier")]
        [fun   (p b)      fwae]  ;; return a function itself as it is a valid value in FWAE
        [app   (f a)      ...]))
```

FWAE: Interpreter

```
; interp: FWAE -> FWAE
(define (interp fwae)
    (type-case FWAE fwae
        [num   (n)        fwae]
        [add   (l r)       (num+ (interp l) (interp r))]
        [sub   (l r)       (num- (interp l) (interp r))]
        [with  (i v e)     (interp (subst e i (interp v)))]
        [id    (s)         (error 'interp "free identifier")]
        [fun   (p b)       fwae]
        [app   (f a)       ...  (interp f) ... (interp a) ... ]))
```

FWAE: Interpreter

```
; interp: FWAE -> FWAE
(define (interp fwae)
    (type-case FWAE fwae
        [num   (n)        fwae]
        [add   (l r)       (num+ (interp l) (interp r))]
        [sub   (l r)       (num- (interp l) (interp r))]
        [with  (i v e)     (interp (subst e i (interp v)))]
        [id    (s)         (error 'interp "free identifier")]
        [fun   (p b)       fwae]
        [app   (f a)       (local [(define ftn (interp f))]
                                  ... (fun-body ftn) ...
                                  ... (fun-param ftn) ...
                                  ...  (interp a) ...)]))
```

FWAE: Interpreter

```
; interp: FWAE -> FWAE
(define (interp fwae)
    (type-case FWAE fwae
        [num   (n)       fwae]
        [add   (l r)      (num+ (interp l) (interp r))]
        [sub   (l r)      (num- (interp l) (interp r))]
        [with  (i v e)    (interp (subst e i (interp v)))]
        [id    (s)        (error 'interp "free identifier")]
        [fun   (p b)      fwae]
        [app   (f a)      (local [(define ftn (interp f))]
                                 (interp (subst (fun-body ftn)
                                                (fun-param ftn)
                                                (interp a))))]))
```

Add and Subtract

```
; num+: FWAE FWAE -> FWAE
(define (num+ x y)
    (num (+ (num-n x) (num-n y))))
; num-: FWAE FWAE -> FWAE
(define (num- x y)
    (num (- (num-n x) (num-n y))))
```

Add and Subtract

```
; num+: FWAE FWAE -> FWAE
(define (num+ x y)
    (num (+ (num-n x) (num-n y))))
; num-: FWAE FWAE -> FWAE
(define (num- x y)
    (num (- (num-n x) (num-n y))))
```

Better:

```
; num-op: (number number -> number) -> (FWAE FWAE -> FWAE)
(define (num-op op)
    (lambda (x y)
        (num (op (num-n x) (num-n y)))))
(define num+ (num-op +))
(define num- (num-op -))
```

* lambda function: https://docs.racket-lang.org/guide/lambda.html

What is a lambda expression? (coding tip)

- Anonymous function
  (https://en.wikipedia.org/wiki/Anonymous_function)
- Pros
  - Code brevity
    - Remove unnecessary loop
    - Reuse a function definition
  - Better performance based on Laziness ← we will learn later.
- Cons
  - Could be slower.
  - Difficult to track function call stack while debugging.
  - Make code difficult to understand.

FWAE subst

```
; subst: FWAE symbol FWAE -> FWAE
(define (subst exp idtf val)
    (type-case FWAE exp
        ...
        [id    (name)      (cond [(equal? name idtf ) val]
                                 [else exp])]
        [app (f arg)       (app (subst f idtf val)
                                (subst arg idtf val))]
        [fun  (id body)  (if (equal? idtf id)
                             exp
                             (fun id (subst body idtf val)))]))
```

A function parameter in definition is equivalent to the binding id in 'with'`

{with {x 3} {fun {x} {+ x y}}} ⇒ (fun 'x (add (id 'x) (id 'y)))
{with {x 3} {fun {y} {+ x y}}} ⇒ (fun 'y (add (num 3) (id 'y)))

FWAE subst

Beware: with the implementation on the previous slide,

```
(subst (with 'y (num 10) (id 'z))
       'z
       (fun 'x (add (id 'x) (id 'y))))
```

⇒

```
(with 'y (num 10) (fun 'x (add (id 'x) (id 'y))))
⇒ (fun 'x (add (id 'x) (num 10))
```

```
(with 'z
      (fun 'x (add (id 'x) (id 'y)))
       (with 'y
             (num 10)
             (id 'z)))
```

FWAE subst

Beware: with the implementation on the previous slide,

```
(subst (with 'y (num 10) (id 'z))
       'z
       (fun 'x (add (id 'x) (id 'y))))
```

```
(with 'z
      (fun 'x (add (id 'x) (id 'y)))
       (with 'y
             (num 10)
             (id 'z)))
```

⇒

(with 'y (num 10) (fun 'x (add (id 'x) (id 'y))))
⇒ (fun 'x (add (id 'x) (num 10))

- which is wrong (as we adopt static scope), but we ignore this problem just for now.
  - Only happens when the original program has free identifiers.
  - The problem disappears *with deferred substitution, anyway*.

Scope

```
{deffun {f p} n}
{with {n 5} {f 10}}
```

- Static scope
  In a language with static scope, the scope of an identifier's binding is a syntactically delimited region.
  The code signals an error.

- Dynamic scope
  In a language with dynamic scope, the scope of an identifier's binding is the entire remainder of the execution during which that binding is in effect.
  The code evaluates to 5.

FWAE subst

Beware: with the implementation on the previous slide,

```
(subst (with 'y (num 10) (id 'z))
       'z
       (fun 'x (add (id 'x) (id 'y))))
⇒
(with 'y (num 10) (fun 'x (add (id 'x) (id 'y))))
```

c.f.

```
(with 'y
      (num 3)
      (with 'z
            (fun 'x (add (id 'x) (id 'y)))
            (with 'y
                  (num 10)
                  (id 'z))))
```

- which is wrong, but we ignore this problem
  - Only happens when the original program has free identifiers
  - The problem disappears *with deferred substitution, anyway*.

Correct evaluation in our language
⇒ (fun 'x (add (id 'x) (id 'y))

Anyway, we've done for FWAE.
But let's think more!

ITP30011 L12
First-class Functions (3)

Lecture12
JC

Where are we now?

AE → WAE → **F1**WAE → F1WAE with def. subst. (END)

→ **F**WAE

→ ***FAE***

→ ***FAE with deferred substitution***

No More 'with'??

```
{with {x 10} x}
```

is the same as

```
{{fun {x} x} 10}
```

No More 'with'

{with {x 10} x}

is the same as

{{fun {x} x} 10}

In general,

{with {<id> <FWAE>$_1$} <FWAE>$_2$}

is the same as

{{fun {<id>} <FWAE>$_2$} <FWAE>$_1$}

No More 'with'

{with {x 10} x}

is the same as

{{fun {x} x} 10}

In general,

{with {<id> <FWAE>$_1$} <FWAE>$_2$}

is the same as

{{fun {<id>} <FWAE>$_2$} <FWAE>$_1$}

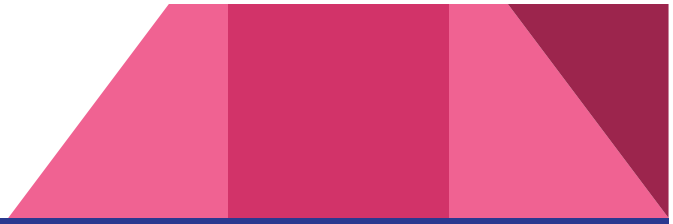
Let's assume

(with '<id> <FWAE>$_1$ <FWAE>$_2$)

⇒ (app (fun '<id> <FWAE>$_2$) <FWAE>$_1$)

FAE: Concrete Syntax

<FAE> ::= <num>
| {+ <FAE> <FAE>}
| {- <FAE> <FAE>}
| {with {<id> <FAE>} <FAE>}
| <id>
| {fun {<id>} <FAE>}
| {<FAE> <FAE>}



FAE: Concrete/Abstract Syntax

<FAE> ::= <num>
| {+ <FAE> <FAE>}
| {- <FAE> <FAE>}
| *{with {<id> <FAE>} <FAE>}*
| <id>
| {fun {<id> <FAE>}
| {<FAE> <FAE>}

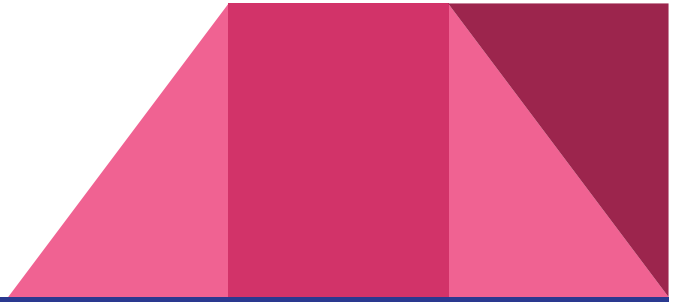
Keep this for concrete syntax!
Remove this for abstract syntax!

- We'll still use 'with' in example code (concrete syntax).
- No more case lines in interp and other functions for 'with'
- No more test cases for interp and other functions using 'with'

Parser Example

(parse {with {x 3} {+ x x}})

⇒



Parser Example

(parse {with {x 3} {+ x x}})

⇒ (app (fun 'x (add (id 'x) (id 'x))) (num 3))

Can you implement the 'parse' function for this?

FAE: Interpreter

```
; interp: FAE -> FAE
(define (interp fae)
   (type-case FAE fae
      [num   (n)      fae]
      [add   (l r)    (num+ (interp l) (interp r))]
      [sub   (l r)    (num- (interp l) (interp r))]
            [with (i v e)  (interp (subst e i (interp v)))]
      [id      (s)    (error 'interp "free identifier")]
      [fun     (p b)  fae]
      [app   (f a)  (local [(define ftn (interp f))]
                           (interp (subst (fun-body ftn)
                                          (fun-param ftn)
                                          (interp a))))]))
```

This still has an issue (dynamic scope) like the following case:

{with {z {fun {x} {+ x y}}} {with {y 10} z}} ⇒

(with 'z (fun 'x (add (id 'x) (id 'y)))
                (with 'y (num 10) (id 'z))) ⇒ ??? in FAE

F1WAE Interpreter with Defrdsub

```
; interp : F1WAE list-of-FucDef DefrdSub -> number
(define (interp f1wae fundefs ds)
 (type-case F1WAE f1wae
  [num  (n)      n]
  [add  (l r)    (+ (interp l fundefs ds) (interp r fundefs ds))]
  [sub  (l r)    (- (interp l fundefs ds) (interp r fundefs ds))]
  [with (i v e)  (interp e fundefs (aSub i (interp v fundefs ds) ds))]
  [id     (s)    (lookup s ds)]
  [app  (f a)    (local
                   [(define a-fundef (lookup-fundef f fundefs))]
                   (interp (fundef-body a-fundef)
                      fundefs
                     (aSub (fundef-arg-name a-fundef)
                           (interp a fundefs ds)
                           (mtSub))
                     ))])))

(test (interp (parse '{f 1}) (list (parse-fd '{deffun (f x) {+ x 3}})) (mtSub)) 4)
```

FAE: (incomplete) Interpreter with Deferred Substitution

```
; interp: FAE DefrdSub > FAE
(define (interp fae ds)
   (type-case FAE fae
      [num   (n)      fae]
      ...
      [id  (s)     (lookup s ds)]
      [fun     (p b)    fae]
      [app    (f a)  (local ([define ftn (interp f ds)])
                            (interp (fun-body ftn)
                            (aSub (fun-param ftn)
                                  (interp a ds)
                                   ds)
                 ))])))
```

Why not (mtSub)? ⇒ In FAE, a function is a value. So when we interpet a function body, we need to substitute identifiers in the body. Values of identifiers of the body are in ds.

(interp (parse '{with {x **3**} {with {f {fun {y} {+ x y}}} {with {x 5} {f 4}}}}}) (mtSub))
⇒ Evaluated as (num 9)

Dynamic scope again??

FAE: (incomplete) Interpreter with Deferred Substitution

```
; interp: FAE DefrdSub > FAE
(define (interp fae ds)
   (type-case FAE fae
      [num   (n)      fae]
      ...
      [id  (s)     (lookup s ds)]
      [fun     (p b)    fae]
      [app    (f a)  (local ([define ftn (interp f ds)])
                            (interp (fun-body ftn)
                            (aSub (fun-param ftn)
                                  (interp a ds)
                                   ds))))]))
```

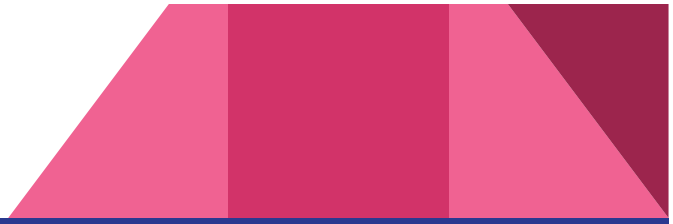
This must be 3 since it is a free id in 'fun {y}...' and x is defined its outer 'with' expression but...

(interp (parse '{with {x **3**} **{**with {f {fun {y} {+ x y}}} **{**with {x **5**} {f 4}**}}**}) (mtSub))

⇒ But evaluated as (num 9) **Dynamic scope again??**
How can we solve this?

FAE with Deferred Substitution

(interp (parse '{with {y 10} {fun {x} {+ y x}}})) [ ]



FAE with Deferred Substitution

(interp (parse '{with {y 10} {fun {x} {+ y x}}})) [ ]

⇒

(interp (parse '{fun {x} {+ y x}})) [y=10]

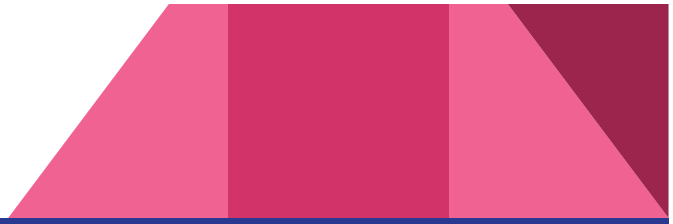
FAE with Deferred Substitution

(interp (parse '{with {y 10} {fun {x} {+ y x}}})) [ ]

⇒

(interp (parse '{fun {x} {+ y x}})) [y=10]

(interp (parse '{{fun {y} {fun {x} {+ y x}}} 10})) [ ]



FAE with Deferred Substitution

(interp (parse '{with {y 10} {fun {x} {+ y x}}})) [ ]

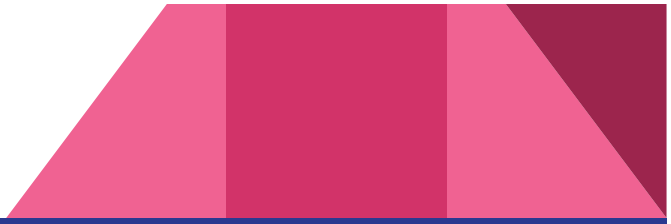
⇒

(interp (parse '{fun {x} {+ y x}})) [y=10]

(interp (parse '{{fun {y} {fun {x} {+ y x}}} 10})) [ ]

⇒

(interp (parse '{fun {x} {+ y x}})) [y=10]



FAE with Deferred Substitution

(interp (parse '{{with {y 10} {fun {x} {+ y x}}}
{with {y 7} y}})) [ ]

Argument expression:

(interp (parse '{with {y 7} y})) [ ]

⇒ (interp (parse 'y)) [y=7]

⇒ 7

FAE with Deferred Substitution

(interp (parse '{{with {y 10} {fun {x} {+ y x}}}
{with {y 7} y}})) [ ]

Argument expression:

(interp (parse '{with {y 7} y})) [ ]

⇒ (interp (parse 'y)) [y=7]

⇒ 7

Function expression:

(interp (parse '{{with {y 10} {fun {x} {+ y x}}}
{with {y 7} y}})) [ ]

⇒ (interp (parse '{fun {x} {+ y x}})) [y=10]

⇒ ??? ⇐ We need a new way to represent this function and this cache together.

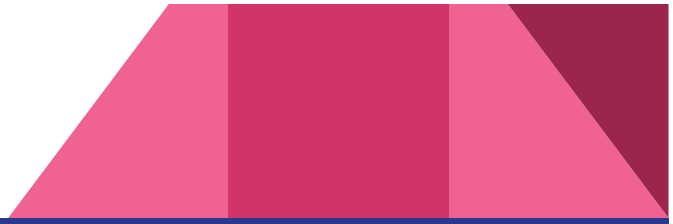
FAE values

Any bound ids which are not a parameter of a function need to be kept in its substitution cache with its corresponding value so that we can avoid dynamic scope and we **will not forget the pending substitution** for the function.

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

(define-type DefrdSub
   [mtSub]
   [aSub (name symbol?) (value FAE-Value?) (ds DefrdSub?)])

(test (interp (parse '{with {y 10} {fun {x} {+ y x}}}) (mtSub))
      (closureV …))
```



FAE values

Any bound ids (e.g., y) which are not a parameter of a function need to be kept in its substitution cache with its corresponding value (e.g., 10) so that we can avoid dynamic scope and we **will not forget the pending substitution** for the function.

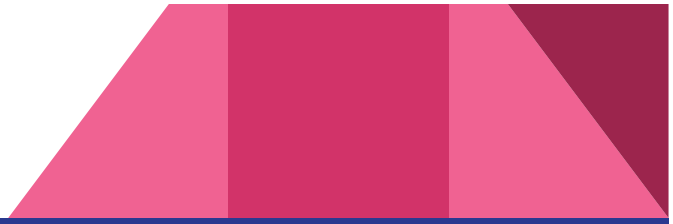
```
(define-type FAE-Value
   [numV       (n number?)]
   [closureV   (param symbol?) (body FAE?) (ds DefrdSub?)])

(define-type DefrdSub
   [mtSub]
   [aSub (name symbol?) (value FAE-Value?) (ds DefrdSub?)])

(test (interp (parse '{with {y 10} {fun {x} {+ y x}}}) (mtSub))
      (closureV 'x (add (id 'y) (id 'x))
                   (aSub 'y (numV 10) (mtSub))))
```

Continuing Evaluation

Function: {fun {x} {+ y x}} [y=10]
Argument: 7



Continuing Evaluation

Function: {fun {x} {+ y x}} [y=10]
Argument: 7

To apply, interpret the function body with the given argument:
(interp (parse '...))

Continuing Evaluation

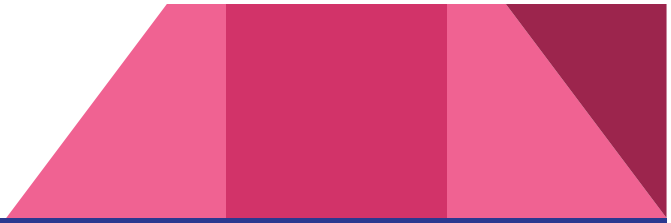
Function: {fun {x} {+ y x}} [y=10]
Argument: 7

To apply, interpret the function body with the given argument:
(interp (parse '{+ y x})) [...]

Continuing Evaluation

Function: {fun {x} {+ y x}} [y=10]
Argument: 7

To apply, interpret the function body with the given argument:
(interp (parse '{+ y x})) [x=7 y=10]



FAE Interpreter with Deferred Substitution

```
; interp: FAE DefrdSub -> FAE-Value
(define (interp fae ds)
   (type-case FAE fae
      [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    (s)     (lookup s ds)]
      [fun   (p b)   …]
      [app   (f a)   …]))
```

FAE Interpreter with Deferred Substitution

```
; interp: FAE DefrdSub -> FAE-Value
(define (interp fae ds)
   (type-case FAE fae
      [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    (s)     (lookup s ds)]
      [fun   (p b)   (closureV p b ds)]
      [app   (f a)   …]))
```

FAE Interpreter with Deferred Substitution

```
; interp: FAE DefrdSub -> FAE-Value
(define (interp fae ds)
   (type-case FAE fae
      [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    (s)     (lookup s ds)]
      [fun   (p b)   (closureV p b ds)]
      [app   (f a)   … (interp f ds)
                     … (interp a ds) …]))
```

FAE Interpreter with Deferred Substitution

```
; interp: FAE DefrdSub -> FAE-Value
(define (interp fae ds)
   (type-case FAE fae
      [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    (s)     (lookup s ds)]
      [fun   (p b)   (closureV p b ds)]
      [app   (f a)   (local [(define f-val (interp f ds))
                             (define a-val (interp a ds))]
                       ...)]))
```

FAE Interpreter with Deferred Substitution

```
; interp: FAE DefrdSub -> FAE-Value
(define (interp fae ds)
   (type-case FAE fae
      [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    (s)     (lookup s ds)]
      [fun   (p b)   (closureV p b ds)]
      [app   (f a)   (local [(define f-val (interp f ds))
                             (define a-val (interp a ds))]
                       (interp (closureV-body f-val)
                               ...))]))
```

FAE Interpreter with Deferred Substitution

```
; interp: FAE DefrdSub -> FAE-Value
(define (interp fae ds)
   (type-case FAE fae
      [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    (s)     (lookup s ds)]
      [fun   (p b)   (closureV p b ds)]
      [app   (f a)   (local [(define f-val (interp f ds))
                             (define a-val (interp a ds))]
                       (interp (closureV-body f-val)
                               (aSub (closureV-param f-val)
                                     a-val
                                     ...)))]))
```

```
(parse '{with {y 10} {fun {x} {+ y x}}}) ; ⇒ (app (fun 'y (fun 'x (add (id 'y) (id 'x)))) (num 10))
(test (interp (parse '{with {y 10} {fun {x} {+ y x}}}) (mtSub))
      (closureV 'x (add (id 'y) (id 'x))       (aSub 'y (numV 10) (mtSub))))
```

FAE Interpreter with Deferred Substitution

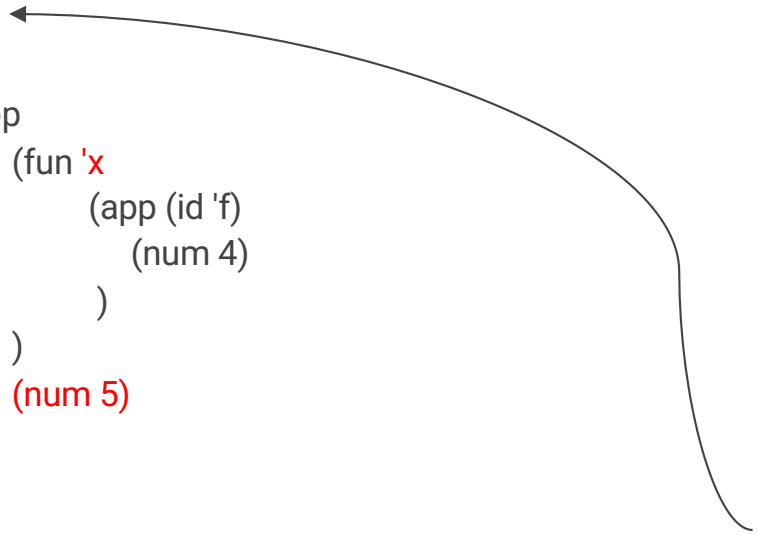
```
; interp: FAE DefrdSub -> FAE-Value
(define (interp fae ds)
   (type-case FAE fae
      [num   (n)     (numV n)]
      [add   (l r)   (num+ (interp l ds) (interp r ds))]
      [sub   (l r)   (num- (interp l ds) (interp r ds))]
      [id    (s)     (lookup s ds)]
      [fun   (p b)   (closureV p b ds)]
      [app   (f a)   (local [(define f-val (interp f ds))
                             (define a-val (interp a ds))]
                       (interp (closureV-body f-val)
                               (aSub (closureV-param f-val)
                                     a-val
                                     (closureV-ds f-val))))]))
```

(interp (parse '{with {**x 3**} {with {f {fun {y} {+ x y}}} {with {x 5} {f 4}}}}) (mtSub))
(interp (app (fun **'x** (app (fun 'f (app (fun 'x (app (id 'f) (num 4))) (num 5))) (fun 'y (add (id 'x) (id 'y))))) **(num 3)**) (mtSub))
⇒ Evaluated as (numV 7)

FAE Interpreter with Deferred Substitution

```
(app
   (fun 'x
        (app
            (fun 'f
                 (app
                      (fun 'x
                           (app (id 'f)
                                (num 4)
                           )
                      )
                      (num 5)
                 )
            )

            (fun 'y
                 (add (id 'x) (id 'y))    ⇒ (closureV 'y (add (id 'x) (id 'y)) (aSub 'x (numV 3) (mtSub)))
            )
        )
    )
    (num 3)
```

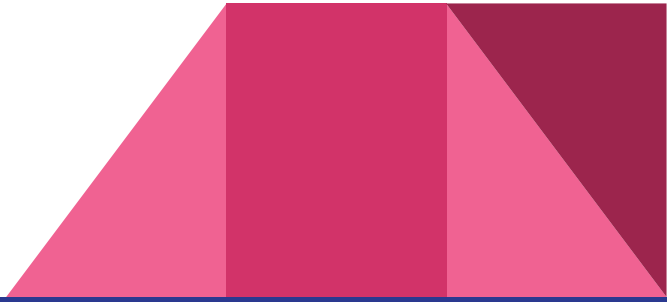


(interp (parse '{with {x 3} {with {f {fun {y} {+ x y}}} {with {x 5} {f 4}}}}) (mtSub))
(interp (app (fun 'x (app (fun 'f (app (fun 'x (app (id 'f) (num 4))) (num 5))) (fun 'y (add (id 'x) (id 'y))))) (num 3)) (mtSub))
⇒ Evaluated as (numV 7)

Environments

```
; interp: FAE DefrdSub -> FAE-Value
(define (interp fae ds)
...
```

http://cs.brown.edu/courses/cs173/2012/book/From_Substitution_to_Environments.html



Topics we cover and schedule (tentative)

- Racket tutorials (L2,3)
- Modeling languages (L4,5)
- Interpreting arithmetic (L5)
- Language principles
  - Substitution (L6-7)
  - Function (L8)
  - Deferring Substitution (L9)
  - First-class Functions (L10-L12)
  - Laziness (L13,14)
  - Recursion (L15,16)
  - Mutable data structures (L17,18,19,20)
  - Variables (L21,22)
  - Continuations (L23-26)
- Guest Video Lecture (L27)

TODO

Read Chapter 8. Implementing Laziness

JC

jcnam@handong.edu

https://lifove.github.io

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