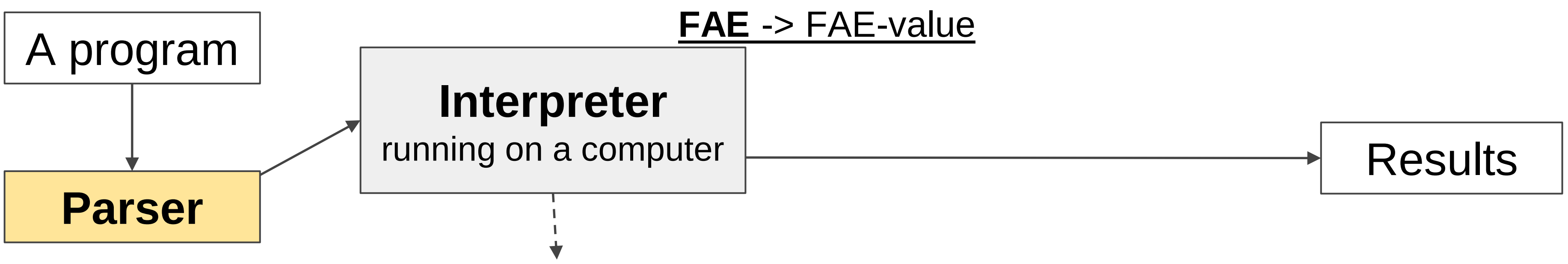
ITP20005
Laziness

Lecture13
JC

Big Picture (modeling languages: substitution)



A program

Parser

s-exp -> **FAE**

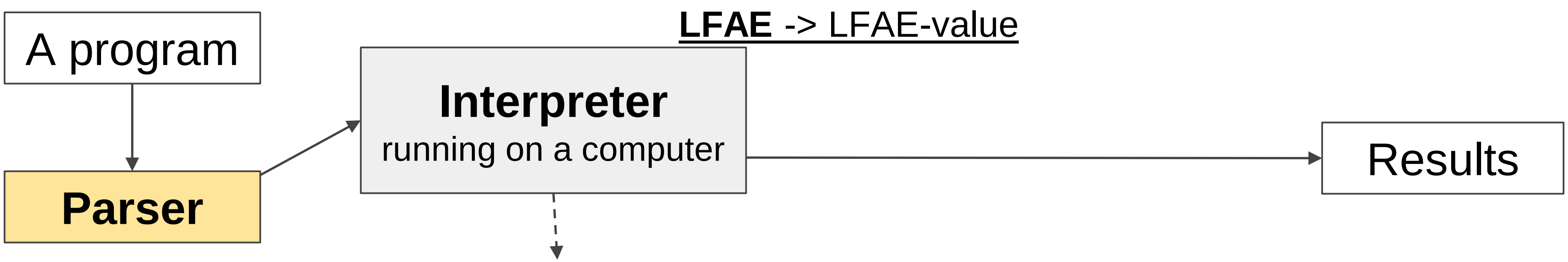
Interpreter
running on a computer

FAE -> FAE-value

Results

Interpreter now will support
(1) Substitution
(2) Function
(3) Deferring Substitution
(4) First-class Functions

Big Picture (modeling languages: substitution)



A program

Parser

Interpreter
running on a computer

Results

LFAE -> LFAE-value

s-exp -> **LFAE**

Interpreter now will support
(1) Substitution
(2) Function
(3) Deferring Substitution
(4) First-class Functions
(5) Laziness

Racket vs. Algebra

In Racket, we have a specific order for evaluating expressions.

(+ (* 4 3) (- 8 7) ⇒ (+ 12 (- 8 7)) ⇒ (+ 12 1)

In Algebra, order does not matter.

$$(4 \cdot 3) + (8 - 7) \Rightarrow 12 + (8 - 7) \Rightarrow 12 + 1$$

or:

$$(4 \cdot 3) + (8 - 7) \Rightarrow (4 \cdot 3) + 1 \Rightarrow 12 + 1$$

Algebra Shortcuts

In Algebra, if we see:

$$f(x,y) = x$$

$$g(z) = \ldots$$

$$f(17,g(g(g(g(g(18))))))$$

then we can go straight to:

$$17$$

because the result of all the g calls will not be used.

Lazy Evaluation

- Languages like Racket, Java, and C are called eager.
  - An expression is evaluated when it is encountered.
- Languages that avoid unnecessary work are called lazy.
  - An expression is evaluated only if its result is needed.
  - What we did in the previous slide is lazy evaluation.
  - Efficient!

Another example

```
{with {x {+ 4 {+ 5 {+ 7 8}}}}
      {with {y {+ 9 10}}
            {with {z y}
                  {with {x 4}
                        z}}}}
```

Another example: try substitution

```
{with {x {+ 4 {+ 5 {+ 7 8}}}}
      {with {y {+ 9 10}}
            {with {z y}
                  {with {x 4}
                        z}}}}
```

Another example: try deferred substitution

```
{with {x {+ 4 {+ 5 {+ 7 8}}}}
          {with {y {+ 9 10}}
                    {with {z y}
                              {with {x 4}
                                        z}}}}
```

Another example: better way?

```
{with {x {+ 4 {+ 5 {+ 7 8}}}}
     {with {y {+ 9 10}}
          {with {z y}
               {with {x 4}
                    z}}}}
```

Lazy Evaluation

- Languages like Scheme, Java, and C are called eager.
    - An expression is evaluated when it is encountered.
- Languages that avoid unnecessary work are called lazy.
    - An expression is evaluated only if its result is needed.
    - What we did in the previous slide is lazy evaluation.
    - Efficient!

New Language that supports lazy evaluation: LFAE

LFAE = Lazy FAE

```
<LFAE> :: = <num>
          | {+ <LFAE> <LFAE>}
          | {- <LFAE> <LFAE>}
          | <id>
          | {fun {<id>} <LFAE>}
          | {<LFAE> <LFAE>}
```

* This grammar is just same as FAE as lazy evaluation is implemented in its interpreter. (No need to change a parser!)

LFAE = Lazy FAE

```
<LFAE> :: = <num>
          | {+ <LFAE> <LFAE>}
          | {- <LFAE> <LFAE>}
          | <id>
          | {fun {<id>} <LFAE>}
          | {<LFAE> <LFAE>}
```

```
{{fun {x} 0} {+ 1 {fun {y} 2}}}
{{fun {x} x} {+ 1 {fun {y} 2}}}
```

LFAE = Lazy FAE

```
<LFAE> :: = <num>
          | {+ <LFAE> <LFAE>}
          | {- <LFAE> <LFAE>}
          | <id>
          | {fun {<id>} <LFAE>}
          | {<LFAE> <LFAE>}
```

{{fun {x} 0} {+ 1 {fun {y} 2}}} ⇒ 0

{{fun {x} x} {+ 1 {fun {y} 2}}} ⇒ error?

Implementing LFAE

Explicitly delay interpretation of argument expressions.

$\{\underbrace{\{\text{fun } \{x\}\ 0\}}_{f}\ \underbrace{\{+\ 1\ \{\text{fun } \{y\}\ 2\}\}}_{a}\} \Rightarrow 0$

```
(define (interp lfae ds)
  (type-case LFAE lfae
    [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-v (interp f ds))                    ???
                              (define arg-v (interp a ds))]                   ???
                         (interp (closureV-body ftn-v)
                                 (aSub (closureV-param ftn-v)
                                       arg-v
                                       (closureV-ds ftn-v))))]))
```

Laziness

"By definition, we should not evaluate the argument expression (until its value is needed); furthermore, to preserve static scope, we should close it over its environment." (Ch 8.1, page 75)

```
{{fun {x} 0} {+ 1 {fun {y} 2}}} ⇒ 0
{{fun {x} x} {+ 1 {fun {y} 2}}} ⇒ error??????
```

Implementing LFAE

Explicitly **delay interpretation of argument expressions**.

$$\{\underbrace{\{\text{fun } \{x\}\ 0\}}_{f}\ \underbrace{\{+\ 1\ \{\text{fun } \{y\}\ 2\}\}}_{a}\} \Rightarrow 0$$

```
(define (interp lfae ds)
   (type-case LFAE lfae
      [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-v (interp f ds))                 ???
                                 (define arg-v (interp a ds))]                ???
                             (interp (closureV-body ftn-v)
                                     (aSub (closureV-param ftn-v)
                                           arg-v
                                           (closureV-ds ftn-v))))]))
```

Implementing LFAE

Explicitly delay interpretation of argument expressions.

```
(define (interp lfae ds)
   (type-case LFAE lfae
      [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-v (interp f ds))                  ???
                                       (define arg-v (exprV a ds))]            new LFAE-Value *
                                 (interp (closureV-body ftn-v)
                                              (aSub (closureV-param ftn-v)
                                                         arg-v
                                                         (closureV-ds ftn-v))))]))
```

* Avoid evaluating 'a' but keep it as it is like ClosureV keeps 'ds'.

Implementing LFAE: LFAE Values

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

Implementing LFAE: LFAE Values

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

DefrdSub vs. Laziness

Implementing LFAE: LFAE Values

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

DefrdSub vs. Laziness

Substitution delayed vs. Evaluation delayed

Both make interpreters efficient!

Implementing LFAE: LFAE Values

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

Short-circuiting vs. Laziness

e_1 && e_2 or e_1 || e_2

Stop right after you know the result. vs. Evaluate only when it is needed.

Cut-off unnecessary computations vs. Delay the whole computation until its result is required.

See some discussions: https://stackoverflow.com/questions/14908548/any-difference-between-lazy-evaluation-and-short-circuit-evaluation/14908813

Forcing Evaluation for Num Operations

```
(define (run sexp ds)
    (interp (parse sexp) ds)) ;; to call parse and interp in one call;)

(run '{{fun {x} {+ 1 x}} 10} (mtSub))

(define (interp lfae ds)
   (type-case LFAE lfae
       ...
       [app (f a)    (local [(define ftn-v (interp f ds))
                             (define arg-v (exprV a ds))]
                        (interp (closureV-body ftn-v)
                                (aSub (closureV-param ftn-v)
                                      arg-v
                                      (closureV-ds ftn-v))))]
```

ftn-v =

arg-v =

Forcing Evaluation for Num Operations

```
(run '{{fun {x} {+ 1 x}} 10} (mtSub))

(define (interp lfae ds)
   (type-case LFAE lfae
      ...
      [fun (p b)  (closureV p b ds)]
      [app (f a)    (local [(define ftn-v (interp f ds))
                            (define arg-v (exprV a ds))]
                         (interp (closureV-body ftn-v)
                                 (aSub (closureV-param ftn-v)
                                       arg-v
                                       (closureV-ds ftn-v))))]

ftn-v     =  (closureV 'x (add (num 1) (id x)) (mtSub))
arg-v     =  (exprV (num 10) (mtSub))
new ds  =  (aSub 'x (exprV (num 10) (mtSub)) (mtSub))
```

Forcing Evaluation for Num Operations

```
(run '{{fun {x} {+ 1 x}} 10} (mtSub))

(define (interp lfae ds)
   (type-case LFAE lfae
      …
      [fun (p b)  (closureV p b ds)]
      [app (f a)    (local [(define ftn-v (interp f ds))
                            (define arg-v (exprV a ds))]
                        (interp (closureV-body ftn-v)
                                (aSub (closureV-param ftn-v)
                                      arg-v
                                      (closureV-ds ftn-v))))]

ftn-v     =  (closureV 'x (add (num 1) (id x)) (mtSub))
arg-v     =  (exprV (num 10) (mtSub))
new ds  =  (aSub 'x (exprV (num 10) (mtSub)) (mtSub))
```

Forcing Evaluation for Num Operations

```
(run '{{fun {x} {+ 1 x}} 10} (mtSub))
(define (interp lfae ds)
   (type-case LFAE lfae
      …
      [fun (p b)  (closureV p b ds)]
      [app (f a)    (local [(define ftn-v (interp f ds))
                                  (define arg-v (exprV a ds))]
                              (interp (closureV-body ftn-v)
                                         (aSub (closureV-param ftn-v)
                                                   arg-v
                                                   (closureV-ds ftn-v))))]
```

```
ftn-v      =  (closureV 'x (add (num 1) (id x)) (mtSub))
arg-v      =  (exprV (num 10) (mtSub))
new ds  =  (aSub 'x (exprV (num 10) (mtSub)) (mtSub))
```

Forcing Evaluation for Num Operations

```
(run '{{fun {x} {+ 1 x}} 10} (mtSub))
(define (interp lfae ds)
  (type-case LFAE lfae
    ...
    [add (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 ftn-v (interp f ds))
                         (define arg-v (exprV a ds))]
                   (interp (closureV-body ftn-v)
                           (aSub (closureV-param ftn-v)
                                 arg-v
                                 (closureV-ds ftn-v))))]
```

```
ftn-v    =  (closureV 'x (add (num 1) (id x)) (mtSub))
arg-v    =  (exprV (num 10) (mtSub))
new ds   =  (aSub 'x (exprV (num 10) (mtSub)) (mtSub))
```

⇒ error: expected numV, got exprV

Forcing Evaluation for Num Operations

```
(run '{{fun {x} {+ 1 x}} 10} (mtSub))
(define (interp lfae ds)
   (type-case LFAE lfae
      …
      [add (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 ftn-v (interp f ds))
                                       (define arg-v (exprV a ds))]
                              (interp (closureV-body ftn-v)
                                          (aSub (closureV-param ftn-v)
                                                    arg-v
                                                    (closureV-ds ftn-v))))]
```

We need to improve the interpreter to solve this error. **HOW?**

```
ftn-v     =  (closureV 'x (add (num 1) (id x)) (mtSub))
arg-v     =  (exprV (num 10) (mtSub))
new-ds  =  (aSub 'x (exprV (num 10) (mtSub)) (mtSub))
⇒ error: expected numV, got exprV
```

Forcing Evaluation for Num Operations

```
(define (num-op op x y)
   (numV (op (numV-n (strict x))
                      (numV-n (strict y)))))
(define (num+ x y) (num-op + x y))
(define (num- x y) (num-op - x y))

; strict: LFAE-Value -> LFAE-Value
(define (strict v)
   (type-case LFAE-Value v
      [exprV (expr ds) (strict (interp expr ds))]
      [else v]))
```

> *"The points where the implementation of a lazy language forces an expression to reduce to a value (if any) are called the* ***strictness points*** *of the language."*

Forcing Evaluation for Application

```
(run '{{fun {f} {f 1}} {fun {x} {+ x 1}}} (mtSub))
; interp: LFAE DefrdSub -> LFAE-Value
(define (interp lfae ds)
   ...
   [app (f a) (local [(define f-val (strict (interp f ds)))
                      (define a-val (exprV a ds))]
                 (interp (closureV-body f-val)
                         (aSub (closureV-param f-val)
                               a-val
                               (closureV-ds f-val))))]))
```

We need to apply 'strict' here.
Why? Let's evaluate this expression.

Forcing Evaluation for Application

```
(run '{{fun {f} {f 1}} {fun {x} {+ x 1}}} (mtSub))
; interp: LFAE DefrdSub -> LFAE-Value
(define (interp lfae ds)
   ...
   [app (f a) (local [(define f-val (strict (interp f ds)))
                      (define a-val (exprV a ds))]
                    (interp (closureV-body f-val)
                            (aSub (closureV-param f-val)
                                  a-val
                                  (closureV-ds f-val))))]))
```

```
f-val    =  (closureV 'f (f (num 1)) (mtSub))
a-val    =  (exprV (fun 'x (add (id 'x) (num 1))) (mtSub))
new ds  =  (aSub 'f (exprV (closureV 'x (add (id 'x) (num 1))) (mtSub)) (mtSub))
```

Forcing Evaluation for Application

```
(run '{{fun {f} {f 1}} {fun {x} {+ x 1}}} (mtSub))
; interp: LFAE DefrdSub -> LFAE-Value
(define (interp lfae ds)
    …
    [app (f a) (local [(define f-val (strict (interp f ds)))
                       (define a-val (exprV a ds))]
                   (interp (closureV-body f-val)
                           (aSub (closureV-param f-val)
                                 a-val
                                 (closureV-ds f-val))))]))
f        = f
a        = (num 1)
ds       =  (aSub 'f (exprV (fun 'x (add (id 'x) (num 1))) (mtSub)) (mtSub))

f-val    = (strict (exprV (closureV 'x (add (id 'x) (num 1))) (mtSub)))
         = (closureV 'x (add (id 'x) (num 1)) (mtSub))
a-val    = (exprV (num 1) (mtSub))
new-ds   = …
```

ITP20005

Laziness (2)

Lecture14
JC

Redundant Evaluation

```
{{fun {x} {+ {+ x x} {+ x x}}}
   {- {+ 4 5} {+ 8 9}}}
```

How many times is {+ 8 9} evaluated?

Since the result is always the same, we'd like to evaluate {- {+ 4 5} {+ 8 9}} at most once.

```
(define (interp lfae ds)
   (type-case LFAE lfae
      …
      [add (l r)    (num+ (interp l ds) (interp r ds))]
      …
```

Boxes in DrRacket

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

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

- box: (define answer (box 0))
- set-box!: (set-box! answer 42)
- unbox: (unbox answer)
- box/c: (box/c number?)

⇒ for dealing with contract for a type in a box.

Caching Strict Results

```
(define-type LFAE-Value
      [numV        (n number?)]
      [closureV    (param symbol?) (body LFAE?) (ds DefrdSub?)]
      [exprV        (expr LFAE?) (ds DefrdSub?)
                    (value (box/c (or/c false LFAE-Value?)))])

; strict: LFAE-Value -> LFAE-Value
(define (strict v)
   (type-case LFAE-Value v
      [exprV (expr ds v-box)
                (strict (interp expr ds))
                ]
      [else v]))
```

Caching Strict Results

```
(define-type LFAE-Value
      [numV        (n number?)]
      [closureV    (param symbol?) (body LFAE?) (ds DefrdSub?)]
      [exprV        (expr LFAE?) (ds DefrdSub?)
                     (value (box/c (or/c false LFAE-Value?)))])

; strict: LFAE-Value -> LFAE-Value
(define (strict v)
   (type-case LFAE-Value v
      [exprV (expr ds v-box)
                (if (not (unbox v-box))     ;; box contains #f? Then evaluate expr as
needed.
                    (local [(define v (strict (interp expr ds)))]
                       (begin (set-box! v-box v)
                                 v))      ;; return v after evaluating it.
                    (unbox v-box))] ;; just unbox to return the value that was already evaluated once.
```

A 'local' block is for definitions + body. A 'begin' block for everything in body (but can't be used in certain context).

Fix up Interpreter

```
; interp: LFAE DefrdSub -> LFAE-Value
(define (interp lfae ds)
  …
  [app (f a)
       (local [(define f-val (strict (interp f ds)))
                  (define a-val (exprV a ds (__________)))]
                (interp (closureV-body f-val)
                       (aSub (closureV-param f-val)
                             a-val
                             (closureV-ds f-val))))]))
```

Fix up Interpreter

```
; interp: LFAE DefrdSub -> LFAE-Value
(define (interp lfae ds)
  …
  [app (f a)
        (local [(define f-val (strict (interp f ds)))
                    (define a-val (exprV a ds (box #f)))]
                 (interp (closureV-body f-val)
                         (aSub (closureV-param f-val)
                               a-val
                               (closureV-ds f-val))))]))
```

LFAE = Lazy FAE

```
<LFAE> :: = <num>
          | {+ <LFAE> <LFAE>}
          | {- <LFAE> <LFAE>}
          | <id>
          | {fun {<id>} <LFAE>}
          | {<LFAE> <LFAE>}
```

{{fun {x} 0} {+ 1 {fun {y} 2}}} ⇒ 0

{{fun {x} x} {+ 1 {fun {y} 2}}} => error?

LFAE = Lazy FAE

```
<LFAE> :: = <num>
          | {+ <LFAE> <LFAE>}
          | {- <LFAE> <LFAE>}
          | <id>
          | {fun {<id>} <LFAE>}
          | {<LFAE> <LFAE>}
```

```
{{fun {x} 0} {+ 1 {fun {y} 2}}} ⇒ 0
{{fun {x} x} {+ 1 {fun {y} 2}}}
   ⇒ (exprV (add (num 1) (fun 'y (num 2))) (mtSub) '#&#f)
   ⇒ Error

{{fun {x} {+ x x}} {+ 1 {fun {y} 2}}} ⇒ Error
```

Laziness

"By definition, we should not evaluate the argument expression (until its value is needed); furthermore, to preserve static scope, we should close it over its environment." (Ch 8.1, page 75)

```
{{fun {x} 0} {+ 1 {fun {y} 2}}} ⇒ 0
{{fun {x} x} {+ 1 {fun {y} 2}}} ⇒ error??????
```

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)
  - Recursion
  - Representation choices
  - Mutable data structures
  - Variables
  - Continuations
  - Garbage collection
  - Semantics
  - Type
- Guest Video Lecture

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

TODO
Read Chapter 8. Recursion

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.