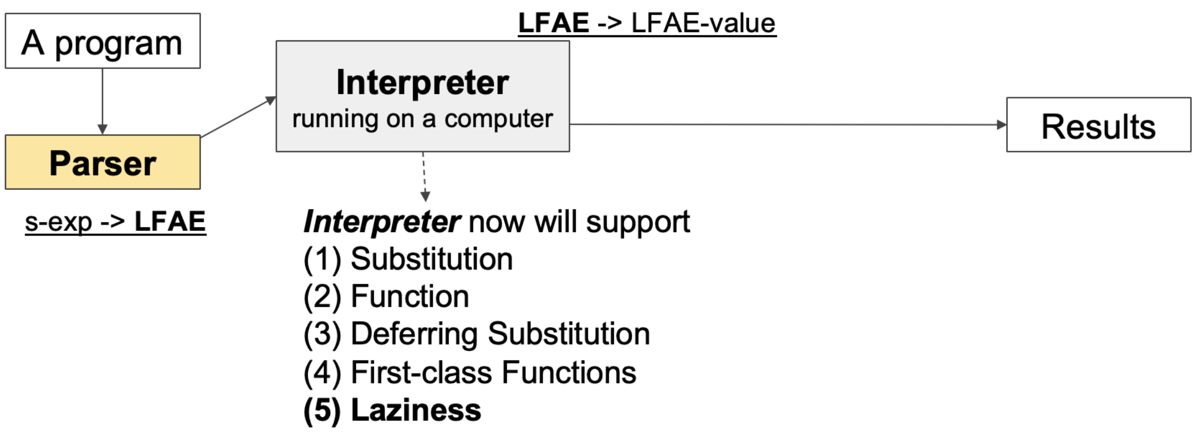
L13. Laziness

유형	강의
복습 여부	✓
Status	Done

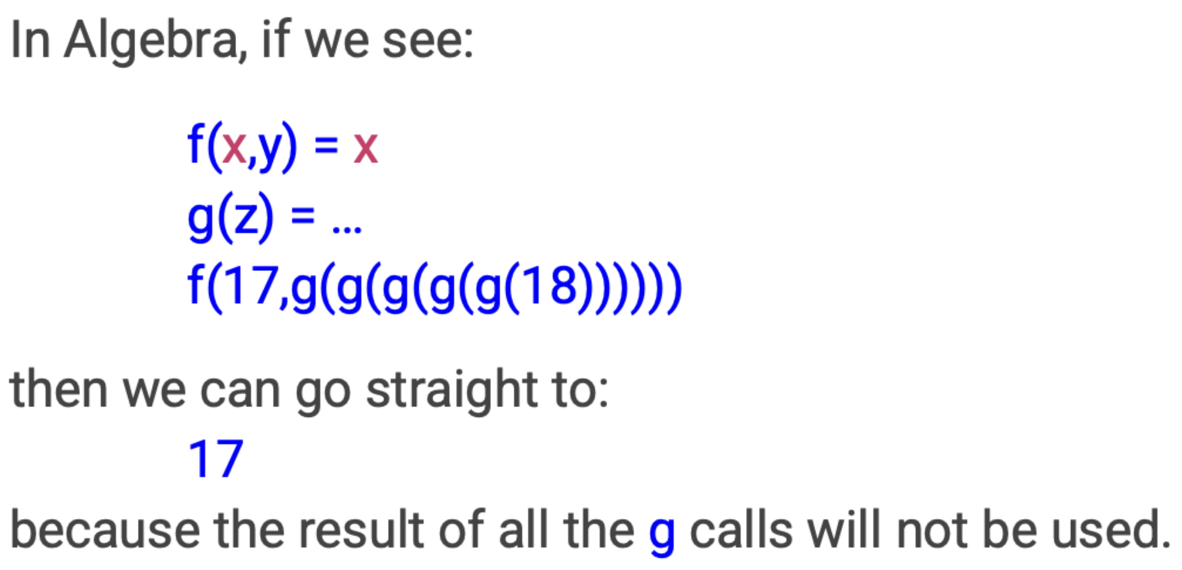


Substitution, function, first-class function are all language concepts.

Deferring substitution is an improved algorithm.

Laziness is also an algorithm to improve efficiency of our language.

Algebra Shortcut



In Algebra, if we see:

```
f(x,y) = x
g(z) = …
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.

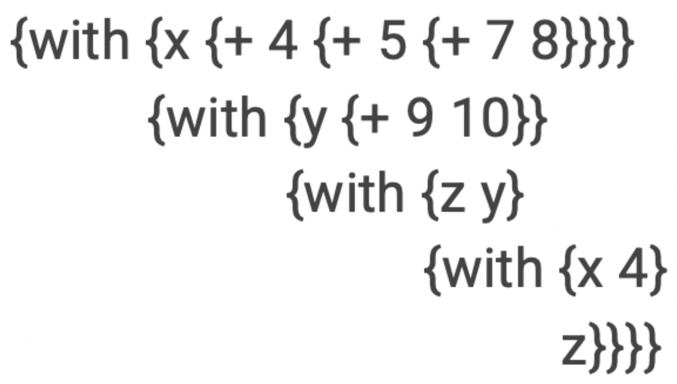
we don't need to evaluate g(g(g(g(g(18))))) part.

Going straight to 17 is called lazy evaluation.

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.
  - For efficiency, evaluate an expression only if its result is needed.

Another Example



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

→ 19

- substitution and deferring substitution inevitably evaluates {x {+ 4 {+ 5 {+ 7 8}}}} part even though x does not have bound identifier
- a better way is lazy evaluation

  = Delay the interpretation of our argument expression

LFAE

a new language that supports lazy evaluation.

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

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

We don’t change our syntax and parser. We just change name of our language.

Q. What does the below example produce?

```
{{fun {x} 0} {+ 1 {fun {y} 2}}}
; without laziness : error
; with laziness : 0

; body of function is 0
; Therefore x is not used in the function body
; So we do not evaluate the argument part, {+ 1 {fun {y} 2}}
```

Q. What does the below example produce ?

```
{{fun {x} x} {+ 1 {fun {y} 2}}}
; without laziness : error
; with laziness : error

; body of function is the argument received through parameter
; {+ 1 {fun {y} 2}} produces error, as {fun {y} 2} is not a n
```

Laziness

Only evaluating what is needed!

We should not evaluate the argument expression until its value is needed.

To preserve static scope, we should close it over its environment.

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

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

The role of exprV

To delay the evaluation of our argument.

exprV data type contains argument expression as it is and deferred substitution cache.

Deferred substitution cache is needed to provide binding information for identifiers in argument expression.

Parser

```
; Parser
(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) (app (fun i (parse e)) (parse 
    [(? symbol?)                (id sexp)]
    [(list 'fun (list p) b)    (fun p (parse b))]
    [(list f a)                 (app (parse f) (parse a))]
    [else                       (error 'parse "bad syntax: ~a"
```

Interpreter

cf. test case 편하게 실행하기 위한 함수

```
; run: sexp -> LFAE-Value
; purpose: to run parse and interp in one queue.
(define (run sexp ds)
     (interp (parse sexp) ds))
```

▼ 시도 1

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

```
; lookup: symbol DefrdSub -> number
(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "free identifier")]
    [aSub (i v saved) (if (symbol=? i name)
                          v
                          (lookup name saved))]))
```

```
(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 (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))
                         )
                 )
         ]
    )
  )
```

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

```
numV-n: contract violation
  expected: numV?
  given: (exprV (num 10) (mtSub))
```

ftn-v = (closureV 'x (add (num 1) (id x)) (mtSub))

arg-v = (exprV (num 10) (mtSub))

(interp (add (num 1) (id ‘x) (aSub 'x (exprV (num 10) (mtSub)) (mtSub)))

(num+

(interp (num 1) (aSub 'x (exprV (num 10) (mtSub)) (mtSub)))

(interp (id ‘x) (aSub 'x (exprV (num 10) (mtSub)) (mtSub)))

)

(num+ (numV 1) **(exprV (num 10) (mtSub))**)

→ expected numV type, but got exprV type.

▼ 수정 1

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

(define num+ (num-op +))
(define num- (num-op -))

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

The role of strict

> 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.

When evaluating the argument in the body of our function, we must unwrap the exprV type by using the strict function. Extract actual value from exprV type.

```
; interpreter
; in app (f a), avoid evaluating 'a' but keep it as it is
(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 (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)
                               )
                         )
                 )
         ]
    )
  )
```

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

```
(numV 11)
```

▼ 달라진 점

ftn-v   = (closureV 'x (add (num 1) (id x)) (mtSub))

arg-v   = (exprV (num 10) (mtSub))

(interp (add (num 1) (id ‘x) (aSub 'x (exprV (num 10) (mtSub)) (mtSub)))

(num+

    (interp (num 1) (aSub 'x (exprV (num 10) (mtSub)) (mtSub)))

    (interp (id ‘x) (aSub 'x (exprV (num 10) (mtSub)) (mtSub)))

)

(num+

    (numV 1)

    (exprV (num 10) (mtSub))

)

(numV (+ (numV-n (strict (numV 1)) (numV-n (strict (exprV (num 10) (mtSub))))))

(numV (+ (numV-n (numV 1)) (numV-n (strict (interp (num 10) (mtSub))))))

(numV (+ (numV-n (numV 1)) (numV-n (strict (numV 10)))))

(numV (+ (numV-n (numV 1)) (numV-n (numV 10))))

(numV (+ 1 10)

(numV 11)

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

```
closureV-body: contract violation
expected: closureV?
given: (exprV (fun 'x (add (id 'x) (num 1))) (mtSub))
```

ftn-v = (closureV 'f (app (id 'f) (num 1)) mtSub)

arg-v = (exprV (fun 'x (add (id 'x) (num 1))) (mtSub))

(interp (app (id 'f) (num 1)) (aSub 'f (exprV (fun 'x (add (id 'x) (num 1))) (mtSub)) (mtSub)))

ftn-v

= (interp (id 'f) (aSub 'f (exprV (fun 'x (add (id 'x) (num 1))) (mtSub)) (mtSub)))

= (lookup 'f (aSub 'f (exprV (fun 'x (add (id 'x) (num 1))) (mtSub)) (mtSub)))

= (exprV (fun 'x (add (id 'x) (num 1))) (mtSub))

arg-v

= (exprV (num 1) (aSub 'f (exprV (fun 'x (add (id 'x) (num 1))) (mtSub)) (mtSub)))

(interp .. unable to continue from here because ftn-v is not ClosureV type

▼ 수정 2

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

(define num+ (num-op +))
(define num- (num-op -))

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

```
; interpreter
; in app (f a), avoid evaluating 'a' but keep it as it is
; in app (f a), also need to apply strict function when in
(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 (s) (lookup s ds)]
    [fun (p b) (closureV p b ds)]
    [app (f a) (local
                 [(define ftn-v (strict (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)
                               )
                         )
                 )
         ]
    )
  )
```

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

```
(numV 11)
(numV 2)
```

▼ 달라진 점

ftn-v
=
(strict (interp (fun 'f (app (id 'f) (num 1))) (mtSub)))
= (strict (closureV 'f (app (id 'f) (num 1)) mtSub))
=
(closureV 'f (app (id 'f) (num 1)) mtSub)

arg-v
= (exprV (fun 'x (add (id 'x) (num 1))) (mtSub))

(interp (app (id 'f) (num 1)) (aSub 'f (exprV (fun 'x (add (id 'x) (num 1)))
(mtSub))
(mtSub)))

ftn-v
=
**(strict (interp (id 'f) (aSub 'f (exprV (fun 'x (add (id 'x) (num 1))) (mtSub))
(mtSub))))**
= (strict (lookup 'f (aSub 'f (exprV (fun 'x (add (id 'x) (num 1))) (mtSub))
(mtSub))))
= (strict (exprV (fun 'x (add (id 'x) (num 1))) (mtSub)))
= (strict (interp (fun 'x (add (id 'x) (num 1))) (mtSub)))
= (strict (closureV 'x (add (id 'x) (num 1)) (mtSub)))
=
(closureV 'x (add (id 'x) (num 1)) (mtSub))

arg-v
(exprV (num 1) (aSub 'f (exprV (fun 'x (add (id 'x) (num 1))) (mtSub))
(mtSub)))

(interp **(add (id 'x) (num 1)**) (aSub **'x** (exprV (num 1) (aSub 'f (exprV (fun 'x
(add (id 'x) (num 1))) (mtSub)) (mtSub))) **(mtSub)**))

…

(numV (+ (numV-n (numV 1)) (numV-n (numV 1))))
(numV 2)

헷갈리지 말아야 할 개념

1. **DefrdSub vs Laziness**

DefrdSub	**Laziness**
Substitution delayed	Evaluation delayed

Both Make interpreters efficient.

2. **Short-circuiting vs Laziness**

Short-circuiting	**Laziness**
Stop right after you know the result	Evaluate only when it’s needed
Cut off unnecessary computations	Delay the whole computation until its result is required