ITP20005 L8
Introduction to Functions

Lecture08
JC

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)

Q&A

- The 'with' part of the function, subst.

```
; [contract] subst: WAE symbol number -> WAE
(define (subst wae idtf val)
        (type-case WAE wae
                [num    (n)      wae]
                [add    (l r)     (add (subst l idtf val) (subst r idtf val))]
                [sub    (l r)     (sub (subst l idtf val) (subst r idtf val))]
                [with   (i v e)   (with i (subst v idtf val) (if (symbol=? i idtf) e
                                  (subst e idtf val)))]
                [id       (s)      (if (symbol=? s idtf) (num val) wae)]))
```

Q&A

- Calling subst in the interpreter.

```
; interp: WAE -> number
(define (interp wae)
    (type-case WAE wae
            [num (n) n]
            [add (l r) (+ (interp l) (interp r))]
            [sub (l r) (- (interp l) (interp r))]
            [with (i v e) (interp (subst e i (interp v)))]
            [id (s)          (error 'interp "free identifier")]))

(test (interp (with 'x (sub (num 7) (num 2))) (add (id 'x) (id 'x   )) 10)
```

Q&A

Target expression for substitution

- The 'with' part of the function, subst.

```
; [contract] subst: WAE symbol number -> WAE
(define (subst wae idtf val)
        (type-case WAE wae
                  [num      (n)        wae]
                  [add      (l r)      (add (subst l idtf val) (subst r idtf val))]
                  [sub      (l r)      (sub (subst l idtf val) (subst r idtf val))]
                  [with     (i v e)    (with i (subst v idtf val)
                                              (if (symbol=? i idtf) e
                                                        (subst e idtf val)))]
                  [id       (s)        (if (symbol=? s idtf) (num val) wae)]))
```

```
; {with {x 10} {...{with {y 17} x}}          ⇒ 10 for x in {with {y 17} x}       ⇒ {with {y 17} 10}
(test (subst (with 'y (num 17) (id 'x)) 'x 10) (with 'y (num 17) (num 10)))
; {with {x 10} {...{with {y x} y}}}          ⇒ 10 for x in {with {y x} y}        ⇒ {with {y 10} y}
(test (subst (with 'y (id 'x) (id 'y)) 'x 10) (with 'y (num 10) (id 'y)))
; {with {x 10} {...{with {y x} x}}}          ⇒ 10 for x in {with {y x} x}        ⇒ {with {y 10} 10}
(test (subst (with 'y (id 'x) (id 'x)) 'x 10) (with 'y (num 10) (num 10)))
```

Q&A

- Calling subst in the interpreter.

```
; interp: WAE -> number
(define (interp wae)
      (type-case WAE wae
               [num (n) n]
               [add (l r) (+ (interp l) (interp r))]
               [sub (l r) (- (interp l) (interp r))]
               [with (i v e) (interp (subst e i (interp v)))]
               [id (s)           (error 'interp "free identifier")]))
```

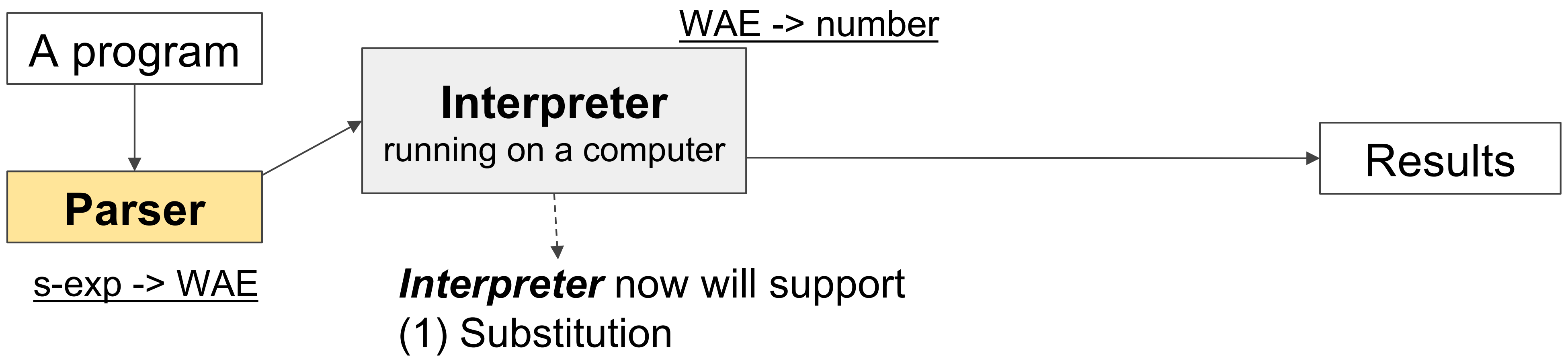
Target expression for substitution

WAE in a value expression

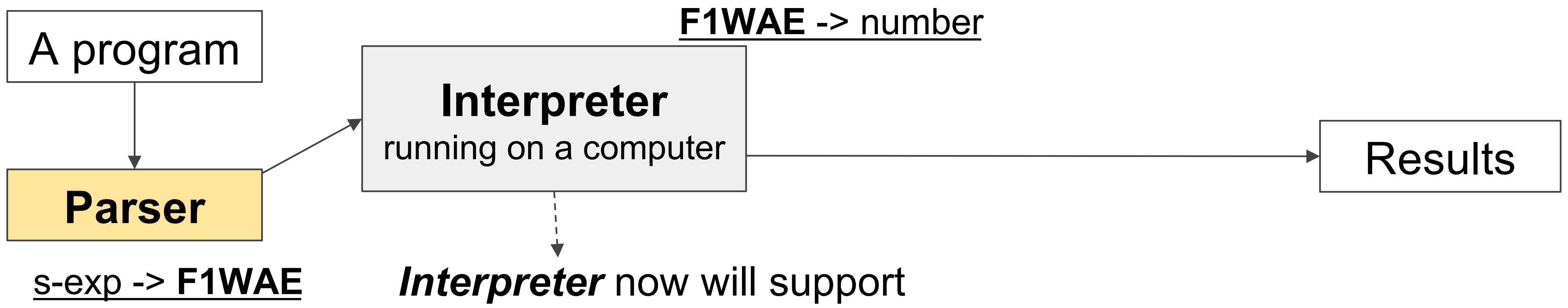
Identifiers (i) in e will be substituted with an actual value from (interp v).

```
(test (interp (with 'x (sub (num 7) (num 2))) (add (id 'x) (id 'x))) 10)
```

Big Picture (modeling languages: substitution)



Big Picture (modeling languages: substitution)



Interpreter now will support
(1) Substitution
(2) **Function**

Expression with an identifier

```
{with {x 5} {+ x 5}}
```

How about?

$f(x) = x + 5$

1 + 5	; f(1)
2 + 5	; f(2)
3 + 5	; f(3)
….	

Parameterized Expression

{+ x 5}

```
1 + 5           ; f(1)
2 + 5           ; f(2)
3 + 5           ; f(3)
....
```

Functions are useful in PL?!

Let's add functions to WAE

We need to define its concrete and abstract syntax.

$\Rightarrow$ Our new language, AE $\longrightarrow$ WAE $\longrightarrow$ F1WAE

Think about simple functions

- *identity(x) = x*
- *twice(x) = x + x*

Think about simple functions

- *identity(x) = x*
- ***twice(x) = x + x***

- AE

  {- 20 {+ 10 10}

  {- 20 {+ 17 17}}

  {- 20 {+ 3 3}}

Think about simple functions

- *identity(x) = x*
- *twice(x) = x + x*

- AE
  ```
  {- 20 {+ 10 10}
  {- 20 {+ 17 17}}
  {- 20 {+ 3 3}}
  ```
- WAE
  ```
  {with {x 10} {- 20 {+ x x}}}
  {with {x 17} {- 20 {+ x x}}}
  {with {x 3} {- 20 {+ x x}}}
  ```

Think about simple functions

- *identity(x) = x*
- *twice(x) = x + x*

- AE
```
{- 20 {+ 10 10}
{- 20 {+ 17 17}}
{- 20 {+ 3 3}}
```
- WAE
```
{with {x 10} {- 20 {+ x x}}}
{with {x 17} {- 20 {+ x x}}}
{with {x 3} {- 20 {+ x x}}}
```
- F1WAE
```
{deffun {identity x}
    x}
```
```
{deffun {twice x}
    {+ x x}}
```

Think about simple functions

- *identity(x) = x*
- *twice(x) = x + x*

- AE
  ```
  {- 20 {+ 10 10}}
  {- 20 {+ 17 17}}
  {- 20 {+ 3 3}}
  ```
- WAE
  ```
  {with {x 10} {- 20 {+ x x}}}
  {with {x 17} {- 20 {+ x x}}}
  {with {x 3} {- 20 {+ x x}}}
  ```
- F1WAE
  ```
  {deffun {identity x}
     x}
  {identity 8}

  17}
  ```
  ```
  {deffun {twice x}
     {+ x x}}
   {twice 10}
      {twice
      {twice
  ```

WAE: Concrete Syntax

```
<WAE> ::= <num>
        | {+ <WAE> <WAE>}
        | {- <WAE> <WAE>}
        | {with {<id> <WAE>} <WAE>}
        | <id>
```

F1WAE: Concrete Syntax

```
<FunDef> ::= {deffun {<id> <id>} <F1WAE>}
<F1WAE> ::= <num>
                                    | {+ <F1WAE> <F1WAE>}
                                    | {- <F1WAE> <F1WAE>}
                                    | {with {<id> <F1WAE>} <F1WAE>}
                                    | <id>
                                    | {<id> <F1WAE>}
```

F1WAE: Concrete Syntax in BNF

```
<FunDef> ::= {deffun {<id> <id>} <F1WAE>}          ← for function definition
<F1WAE> ::= <num>
              | {+ <F1WAE> <F1WAE>}
              | {- <F1WAE> <F1WAE>}
              | {with {<id> <F1WAE>} <F1WAE>}
              | <id>
              | {<id> <F1WAE>}                      ← for function call
```

F1WAE: Concrete Syntax

```
<FunDef> ::= {deffun {<id> <id>} <F1WAE>}        ← for function definition
<F1WAE> ::= <num>
          | {+ <F1WAE> <F1WAE>}
          | {- <F1WAE> <F1WAE>}
          | {with {<id> <F1WAE>} <F1WAE>}
          | <id>
          | {<id> <F1WAE>}                     ← for function call
```

```
{deffun {identity x} x}
{identity 8}

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

How about this??? F1WAE: Concrete Syntax

```
<F1WAE> ::= <num>
          | {+ <F1WAE> <F1WAE>}
          | {- <F1WAE> <F1WAE>}
          | {with {<id> <F1WAE>} <F1WAE>}
          | <id>
          | {<F1WAE> <F1WAE>}              ← function call??
          | {deffun {<id>} <F1WAE>}        ← for function definition
```

F1WAE: Concrete Syntax

```
<FunDef> ::= {deffun {<id> <id>} <F1WAE>}          ← for function definition
<F1WAE> ::= <num>
          | {+ <F1WAE> <F1WAE>}
          | {- <F1WAE> <F1WAE>}
          | {with {<id> <F1WAE>} <F1WAE>}
          | <id>
          | {<id> <F1WAE>}                          ← for function call
```

```
{deffun {identity x} x}
{identity 8}

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

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 F1WAE?)]
          [sub      (lhs F1WAE?) (rhs F1WAE?)]
          [with     (name symbol?) (named-expr F1WAE?) (body F1WAE?)]
          [id                 (name symbol?)]
          [app      (ftn symbol?)      (arg F1WAE?)])
```

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 F1WAE?)]
          [sub      (lhs F1WAE?) (rhs F1WAE?)]
          [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))
```

← Abstract syntax representation of the example code written in our new language.

F1WAE Parser

F1WAE: Concrete Syntax in BNF

```
<FunDef> ::= {deffun {<id> <id>} <F1WAE>}          ← for function definition
<F1WAE> ::= <num>
          | {+ <F1WAE> <F1WAE>}
          | {- <F1WAE> <F1WAE>}
          | {with {<id> <F1WAE>} <F1WAE>}
          | <id>
          | {<id> <F1WAE>}                          ← for function call
```

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 F1WAE?)]
        [sub      (lhs F1WAE?) (rhs F1WAE?)]
        [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))
```

← Abstract syntax representation of the example code written in our new language.

F1WAE Parser

```
; parse-fd: sexp -> FunDef
…

; parse : sexp -> F1WAE
```

F1WAE: Parser

; parse-fd: sexp -> FunDef

```
; parse : sexp -> F1WAE
(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 f a)                           (app f (pars
                    [else                                 (error 'parse "bad syntax: ~a"
sexp)]))
```

F1WAE: Parser

```
; parse-fd: sexp -> FunDef
(define (parse-fd sexp)
          (match sexp
                    [(list 'deffun (list f x) b)    (fundef f x (parse b))]))

; parse : sexp -> F1WAE
(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 f a)                           (app f (pars
                    [else                                 (error 'parse "bad syntax: ~a"
sexp)]))
```

F1WAE: Parser

Function body ← F1WAE

```
; parse-fd: sexp -> FunDef
(define (parse-fd sexp)
          (match sexp
                    [(list 'deffun (list f x) b)    (fundef f x (parse b))]))

; parse : sexp -> F1WAE
(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 f a)                              (app f (parse
                    [else                                    (error 'parse "bad syntax: ~a"
sexp)]))
```

F1WAE Interpreter

```
; interp: F1WAE ?????? -> number
```

F1WAE Interpreter

; interp: F1WAE ?????? -> number

```
(fundef   'identify 'x (id 'x))
(fundef 'twice 'x (add (id 'x) (id 'x)))

(app 'identity (num 8))
(app 'twice (num 10))
```

F1WAE Interpreter

; interp: F1WAE list-of-FuncDef -> number

```
(fundef   'identify 'x (id 'x))
(fundef 'twice 'x (add (id 'x) (id 'x)))

(app 'identity (num 8))
(app 'twice (num 10))
```

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   (x i b)   (interp (subst b x (interp i fundefs))
fundefs)]
                    [id              (s)                (error 'interp "free
identifier")]
                    [app    (f a)     ...]))
```

F1WAE: Interpreter

```
(test (interp (add (num 1) (num 1))
                                                   empty)
       ?)

; 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   (x i b)   (interp (subst b x (interp i fundefs))
fundefs)]
                [id             (s)             (error 'interp "free
identifier")]
```

F1WAE: Interpreter

```
(test (interp (add (num 1) (num 1))
                                        empty)
      2)

; 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   (x i b)   (interp (subst b x (interp i fundefs))
fundefs)]
            [id           (s)             (error 'interp "free
identifier")]
```

F1WAE: Interpreter

```
(test (interp (add (num 1) (num 1))
                                        (list (fundef 'f 'x (add (id 'x) (num 3)))))
          ?)

; 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    (x i b)   (interp (subst b x (interp i fundefs))
fundefs)]
                    [id                (s)                 (error 'interp "free
identifier")]
```

F1WAE: Interpreter

```
(test (interp (add (num 1) (num 1))
                                        (list (fundef 'f 'x (add (id 'x) (num 3)))))
          2)

; 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   (x i b)   (interp (subst b x (interp i fundefs))
fundefs)]
                    [id             (s)                 (error 'interp "free
identifier")]
```

F1WAE: Interpreter

```
(test (interp (app 'f (num 1))
                                        (list (fundef 'f 'x (add (id 'x) (num 3)))))
          ?)

; 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    (x i b)    (interp (subst b x (interp i fundefs))
fundefs)]
                    [id                 (s)                   (error 'interp "free
identifier")]
```

F1WAE: Interpreter

```
(test (interp (app 'f (num 1))
                                    (list (fundef 'f 'x (add (id 'x) (num 3)))))
         4)

; 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   (x i b)   (interp (subst b x (interp i fundefs))
fundefs)]
                  [id              (s)                (error 'interp "free
identifier")]
```

F1WAE: Interpreter

```
(test (interp (app 'f (num 10)
                                        (list (fundef 'f 'x (sub (num 20)

          (app 'twice (id 'x))))
                                                                                (fundef 'twice 'y
(add (id 'y) (id 'y))))))
          ?)
```

```
; 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     (x i b)     (interp (subst b x (interp i fundefs)) fundefs)]
```

F1WAE: Interpreter

```
(test (interp (app 'f (num 10))
                                        (list (fundef 'f 'x (sub (num 20)

          (app 'twice (id 'x))))
                                                                              (fundef 'twice 'y
(add (id 'y) (id 'y)))))
          0)

; 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     (x i b)    (interp (subst b x (interp i fundefs)) fundefs)]
```

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   (x i b)   (interp (subst b x (interp i fundefs))
fundefs)]
                [id             (s)             (error 'interp "free
identifier")]
                [app    (f a)     … (interp a fundefs) …]))
```

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   (x i b)   (interp (subst b x (interp i fundefs))
fundefs)]
                [id             (s)             (error 'interp "free
identifier")]
                [app    (f a)     … (interp a fundefs) …]))

; lookup-fundef: symbol list-of-FunDef -> FunDef
```

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))]
                ...
                [app    (f a)
                        (local
                                [(define a_fundef (lookup-fundef f
fundefs))]
                                (interp (subst (fundef-body
a_fundef)
                                (interp a
fundefs))
```

Get the function definition (body and argument) from the look-up function

(fundef 'f 'x (add (id 'x) (num 3)))

* local: to implement a local logic. https://docs.racket-lang.org/reference/local.html
In our case, we need a local logic for the result expression in a branch of type-case

Lookup

```
; lookup-fundef: symbol list-of-FunDef -> FunDef
(define (lookup-fundef name fundefs)
            ...)
```

Lookup

```
; lookup-fundef: symbol list-of-FunDef -> FunDef
(define (lookup-fundef name fundefs)
          (cond
                    [(empty? fundefs)
                              ...]
                    [else
                              ... (first fundefs)
                              ... (lookup-fundef name (rest fundefs)) ...]))
```

Lookup

```
; lookup-fundef: symbol list-of-FunDef -> FunDef
(define (lookup-fundef name fundefs)
          (cond
                    [(empty? fundefs)
                              (error 'lookup-fundef "unknown function")]
                    [else
                              (if (symbol=? name (fundef-fun-name (first
fundefs)))
                                             (first fundefs)
                                             (lookup-fundef name (rest fundefs)))]))
```

Substitution for F1WAE Interpreter

```
; [contract] subst: F1WAE symbol number -> F1WAE
(define (subst f1wae idtf val)
        (type-case F1WAE f1wae
                [num    (n)              f1wae]
                [add    (l r)            (add (subst l idtf val) (subst r idtf
val))]
                [sub            (l r)            (sub (subst l idtf val) (subst r
idtf val))]
                [with   (i v e)   (with i (subst v idtf val) (if (symbol=? i idtf) e

        (subst e idtf val)))]
                [id             (s)              (if (symbol=? s idtf) (num val)
f1wae)]
                [app    (f a)            (app f   (subst a idtf       ))

; {with {x 1} {fn x}}    <- function call in the body of 'with'.
(subst (app 'fn (id 'x)) 'x 1) ; (app 'fn (num 1))
```

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 PLAI (first edition) Chapter 5. Deferring Substitution

Second edition Ch 6. From Substitution to Environments
http://cs.brown.edu/courses/cs173/2012/book/From_Substitution_to_Environments.html

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.