Why Functions as Values

- Abstraction is easier with functions as values

  ○ **get-ids**

  ○ **filter**, **map**, etc.

- Separate **deffun** form becomes unnecessary

  ○ ```
    {deffun {f x} {+ 1 x}}
    {f 10}
    ⟹
    {with {f {fun {x} {+ 1 x}}}
      {f 10}}
    ```

# FWAE Grammar, Almost

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

FWAE Evaluation

```
10  ⟹  10

{+ 1 2}  ⟹  3

{- 1 2}  ⟹  -1

{with {x 7} {+ x 2}}  ⟹  {+ 7 2}  ⟹  9

y  ⟹   free variable

{fun {x} {+ 1 x}}  ⟹
{fun {x} {+ 1 x}}
```

Result is not always a number!

```
; interp FWAE ... -> FWAE-Value
```

FWAE Evaluation

```
10  ⟹  10

{+ 1 2}  ⟹  3

{- 1 2}  ⟹  -1

{with {x 7} {+ x 2}}  ⟹  {+ 7 2}  ⟹  9

y  ⟹   free variable

{fun {x} {+ 1 x}}  ⟹
{fun {x} {+ 1 x}}

{with {y 10} {fun {x} {+ y x}}}
 ⟹  {fun {x} {+ 10 x}}

{with {f {fun {x} {+ 1 x}}} {f 3}}
 ⟹  {{fun {x} {+ 1 x}} 3}
```

Doesn't match the grammar for **<FWAE>**

FWAE Grammar

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

FWAE Evaluation

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

{{fun {x} {+ 1 x}} 3} ⇒ {+ 1 3} ⇒ 4

{1 2} ⇒ not a function

{+ 1 {fun {x} 10}} ⇒ not a number
```

FWAE Datatype

```
(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 (fun-expr FWAE?)
       (arg-expr FWAE?)])

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

FWAE Datatype

```
(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 (fun-expr FWAE?)
       (arg-expr FWAE?)])


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

FWAE Interpreter

```
; interp : FWAE -> FWAE
(define (interp a-wae)
  (type-case FWAE a-wae
    [num (n) a-wae]
    [add (l r) (num+ (interp l) (interp r))]
    [sub (l r) (num- (interp l) (interp r))]
    [with (bound-id named-expr body-expr)
      (interp (subst body-expr
                     bound-id
                     (interp named-expr)))]
    [id (name) (error 'interp "free variable")]
    [fun (param body-expr)
         a-wae]
    [app (fun-expr arg-expr)
         (local [(define fun-val (interp fun-expr))]
           (interp (subst (fun-body fun-val)
                          (fun-param fun-val)
                          (interp arg-expr))))]))
```

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 : (num num -> num) -> (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 -))
```

FWAE Subst

```
; subst : FWAE symbol -> FWAE
```

Implementation is an exercise for the student

Beware: with the obvious implementation,

```
(subst {with {y 10} z} 'z {fun {x} {+ x y}})
⇒ {with {y 10} {fun {x} {+ x y}}}
```

which is wrong, but we leave this problem to CS 7520

- Only happens when the original program has free variables

- The problem disappears with deferred substitution, anyway

No More With

Compare the **with** and **app** implementations:

```
(define (interp a-wae)
  (type-case FWAE a-wae
    ...
    [with (bound-id named-expr body-expr)
      (interp (subst body-expr
                     bound-id
                     (interp named-expr)))]
    ...
    [app (fun-expr arg-expr)
        (local [(define fun-val (interp fun-expr))]
          (interp (subst (fun-body fun-val)
                         (fun-param fun-val)
                         (interp arg-expr))))]))
```

The **app** case does everything that **with** does

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

```
(test {with {<id> <FWAE>1} <FWAE>2}
      (app (fun '<id> <FWAE>2 ) <FWAE>1 ))
```

FAE Grammar

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

- We'll still use `with` in boxes

- No more case lines in `interp`, etc. for `with`

- No more test cases for `interp`, etc. using `with`

FAE Interpreter

```
; interp : FAE -> FAE
(define (interp a-fae)
  (type-case FAE a-wae
    [num (n) a-fae]
    [add (l r) (num+ (interp l) (interp r))]
    [sub (l r) (num- (interp l) (interp r))]
    [id (name) (error 'interp "free variable")]
    [fun (param body-expr) a-fae]
    [app (fun-expr arg-expr)
         (local [(define fun-val (interp fun-expr))]
            (interp (subst (fun-body fun-val)
                           (fun-param fun-val)
                           (interp arg-expr))))]))
```

FAE with Deferred Substitution

(interp {with {y 10} {fun {x} {+ y x}}})

$\Rightarrow$

(interp {fun {x} {+ y x}})

y = 10

(interp {{fun {y} {fun {x} {+ y x}}} 10})

$\Rightarrow$

(interp {fun {x} {+ y x}})

y = 10

38-41

FAE with Deferred Substitution

```
(interp  {{with {y 10} {fun {x} {+ y x}}}
          {with {y 7} y}}                     )
```

Argument expression:

```
(interp  {with {y 7} y} )
```

$\Rightarrow$

```
(interp  y )  ⟹  7
```
y = 7

Function expression:

```
(interp  {with {y 10} {fun {x} {+ y x}}} )
```

$\Rightarrow$

```
(interp  {fun {x} {+ y x}} )  ⟹  ?
```
y = 10

FAE Values

A function value needs to keep its substitution cache

```
(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 {with {y 10} {fun {x} {+ y x}}})
      (closureV 'x {+ y x}
                (aSub 'y (num 10) (mtSub)))))
```

Continuing Evaluation

Function: {**fun** {**x**} {**+ y x**}}   y = 10

Argument: 7

To apply, interpret the function body with the given argument:

x = 7   y = 10

(**interp** {**+ y x**})

FAE Interpreter with Substitution

```
; interp : FAE DefrdSub -> FAE-Value
(define (interp a-wae ds)
  (type-case FAE a-wae
    [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 (fun-expr arg-expr)
         (local [(define fun-val
                    (interp fun-expr ds))
                  (define arg-val
                    (interp arg-expr ds))]
           (interp (closureV-body fun-val)
                   (aSub (closureV-param fun-val)
                         arg-val
                         (closureV-ds fun-val)))))]))
```