Higher-Order Functions (Part I)

Why Functions as Values

- Abstraction is easier with functions as values
  - abstract over `add` and `sub` cases
  - `filter`, `map`, etc.

- What are objects? Callbacks?

- 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 identifier*

`{fun {x} {+ 1 x}}` ⇒
`{fun {x} {+ 1 x}}`

Result is not always a number!

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

FWAE Evaluation

Let’s think in terms of substitution, for simplicity

```
    {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}  ⇒  expected function

{+ 1 {fun {x} 10}}  ⇒  expected 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-name 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-name 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-Value

```
(define-type FWAE-Value
  [numV (n number?)]
  [funV (param-name symbol?)
        (body FWAE?)])
```

FWAE Interpreter

```
; interp : FWAE? -> FWAE-Value?
(define (interp an-fwae)
  (type-case FWAE an-fwae
    [num (n) (numV n)]
    [add (l r) (num+ (interp l) (interp r))]
    [sub (l r) (num- (interp l) (interp r))]
    [with (name named-expr body)
          (interp (subst body
                         name
                         (interp named-expr)))]
    [id (name) (error 'interp "free identifier")]
    [fun (param-name body)
         (funV param-name body)]
    [app (fun-expr arg-expr)
         (define fun-val (interp fun-expr))
         (interp (subst (funV-body fun-val)
                        (funV-param-name fun-val)
                        (interp arg-expr)))]))
```

Add and Subtract

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

Better:

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

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

FWAE Subst

```
; subst : FWAE? symbol? FWAE-Value? -> FWAE?
(define (subst exp sub-id val)
  (type-case FWAE exp
    ...
    [id (name)
      (cond
        [(equal? name sub-id)
         (type-case FWAE-Value val
           [numV (n) (num n)]
           [funV (param-name body)
                 (fun param-name body)])]
         [else exp]])]
    ...))
```

FWAE Subst

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

FWAE Subst

Beware: with the implementation on the previous slide,

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

- That **y** in the function used to be a free identifier
- and now it’s bound! That can’t be right...
- We’ll see how to fix this

No More With

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

```
(define (interp an-fwae)
  (type-case FWAE an-fwae
    ...
    [with (name named-expr body)
          (interp (subst body
                         name
                         (interp named-expr)))]
    ...
    [app (fun-expr arg-expr)
         (define fun-val (interp fun-expr))
         (interp (subst (funV-body fun-val)
                        (funV-param-name 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}
```

Aside: IIFEs in JavaScript

So let’s just get rid of `with`

Don’t worry, we’ll bring it back

FAE Grammar

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

FAE Interpreter

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

Where to?

- This FAE language will be our "base camp" for most of the rest of the quarter
  - We'll use it to demonstrate concepts
  - We'll extend it in various ways

- Some loose ends we need to tie up
  - Fixing substitution
  - Bringing `with` back
  - Moving to deferred substitution
  - Bringing recursion back (we "lost" it along the way)