Part I

Lexical Addresses and Compilation (Again)

Identifier Address

Suppose that

```
{fun {x} {+ x y}}
```

appears in a program; the body is eventually evaluated:

```
{+ x y}
```

`...`

where will **`x`** be in the substitution?

Answer: always at the beginning:

`x = ...    ...`

Identifier Address

Suppose that

```
{with {y 1} {+ x y}}
```

appears in a program; the body is eventually evaluated:

```
{+ x y}
```

...

where will `y` be in the substitution?

Answer: always at the beginning:

```
y = 1    ...
```

Identifier Address

Suppose that

```
{with {y 1}
  {fun {x} {+ x y}}}
```

appears in a program; the body is eventually evaluated:

```
{+ x y}
```

```
...
```

where will `y` be in the substitution?

Answer: always second:

```
x = ...    y = 1    ...
```

Identifier Address

Suppose that

```
{with {y 1}
  {{fun {x} {- {+ x y} 17}} 88}}
```

appears in a program; the body is eventually evaluated:

```
{+ x y}
```

...

where will **x** and **y** be in the substitution?

Answer: always first and second:

```
x = ...    y = 1    ...
```

Identifier Address

Suppose that

```
{with {y 1}
  {{fun {w} {with {z 9}
              {fun {x} {+ x y}}}}}}
```

appears in a program; the body is eventually evaluated:

```
{+ x y}
```

`...`

where will **x** and **y** be in the substitution?

Answer: always first and fourth:

```
x = ...   z = 9   w = ...   y = 1   ...
```

Identifier Address

Suppose that

```
{with {y {with {r 8} {f {fun {x} r}}}}
  {{fun {w} {with {z 9}
                {fun {x} {+ x y}}}}}}
```

appears in a program; the body is eventually evaluated:

```
...
{+ x y}
```

where will **x** and **y** be in the substitution?

Answer: always first and fourth:

```
x = ...    z = 9    w = ...    y = ...    ...
```

Compiling FAE

```
; compile : FAE ... -> CFAE
```

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

```
(define-type CFAE
  [cnum (n number?)]
  [cadd (lhs CFAE?)
        (rhs CFAE?)]
  [csub (lhs CFAE?)
        (rhs CFAE?)]
  [cat (pos number?)]
  [cfun (body CFAE?)]
  [capp (fun-expr CFAE?)
        (arg-expr CFAE?)])
```

Compile Examples

```
(compile 1 ...) ⇒ 1

(compile {+ 1 2} ...) ⇒ {+ 1 2}

(compile x ...) ⇒ compile: free identifier

(compile {fun {x} x} ...) ⇒ {fun {at 0}}

(compile {fun {y} {fun {x} {+ x y}}} ...)
  ⇒ {fun {fun {+ {at 0} {at 1}}}}

(compile {{fun {x} x} 10} ...)
  ⇒ {{fun {at 0}} 10}
```

Implementing the Compiler

```
; compile : FAE CSubs -> CFAE
(define (compile a-fae cs)
  (type-case FAE a-fae
    [num (n) (cnum n)]
    [add (l r) (cadd (compile l cs)
                     (compile r cs))]
    [sub (l r) (csub (compile l cs)
                     (compile r cs))]
    [id (name) (cat (locate name cs))]
    [fun (param body-expr)
         (cfun (compile body-expr
                        (aCSub param cs)))]
    [app (fun-expr arg-expr)
         (capp (compile fun-expr cs)
               (compile arg-expr cs))]))
```

CFAE Values

Values are still numbers or closures, but a closure doesn't need a parameter name:

```
(define-type CFAE-Value
  [cnumV (n number?)]
  [cclosureV (body CFAE?)
             (subs list?)])
```

CFAE Interpreter

Almost the same as **FAE interp**:

```
; cinterp : CFAE list-of-CFAE-Value -> CFAE-Value
(define (cinterp a-cfae subs)
  (type-case CFAE a-cfae
    [cnum (n) (cnumV n)]
    [cadd (l r) (cnum+ (cinterp l subs) (cinterp r subs))]
    [csub (l r) (cnum- (cinterp l subs) (cinterp r subs))]
    [cat (pos) (list-ref subs pos)]
    [cfun (body-expr)
          (cclosureV body-expr subs)]
    [capp (fun-expr arg-expr)
          (local [(define fun-val
                    (cinterp fun-expr subs))
                  (define arg-val
                    (cinterp arg-expr subs))]
            (cinterp (cclosureV-body fun-val)
                     (cons arg-val
                           (cclosureV-subs fun-val))))]))
```

Part II

Dynamic Scope

Recursion

What if we want to write a recursive function?

```
{with {f {fun {x} {f {+ x 1}}}}
  {f 0}}
```

This doesn't work, because `f` is not bound in the right-hand side of the `with` binding

But by the time that `f` is called, `f` is available...

Dynamic Scope

```
{with {f {fun {x} {f {+ x 1}}}}
  {f 0}}
```

f = `{fun {x} {f {+ x 1}}}`

```
⇒ {f 0}
```

Lexical scope:

x = 0

```
⇒ {f {+ x 1}}
```

Dynamic scope:

x = 0   f = `{fun {x} {f {+ x 1}}}`

```
⇒ {f {+ x 1}}
```

Implementing Dynamic Scope

```
; dinterp : FAE DefrdCache -> FAE-Value
(define (dinterp a-fae ds)
  (type-case FAE a-fae
    [num (n) (numV n)]
    [add (l r) (num+ (dinterp l ds) (dinterp r ds))]
    [sub (l r) (num- (dinterp l ds) (dinterp r ds))]
    [id (name) (lookup name ds)]
    [fun (param body-expr)
         (closureV param body-expr (mtSub))]
    [app (fun-expr arg-expr)
         (local [(define fun-val
                   (dinterp fun-expr ds))
                 (define arg-val
                   (dinterp arg-expr ds))]
           (dinterp (closureV-body fun-val)
                    (aSub (closureV-param fun-val)
                          arg-val
                          ds))))]))
```

Benefits of Dynamic Scope

Dynamic scope looks like a good idea:

- *Seems* to make recursion easier
- Implementation *seems* simple:
  - No closures; change to our interpreter is trivial
  - There's only one binding for any given identifier at any given time
- Supports optional arguments:

```
{with {x 0}
  {with {f {fun {y} {+ x y}}}
    {+ {f 1} ; use default x
       {with {x 3} ; change x to 3
             {f 2}}}}}
```

Drawbacks of Dynamic Scope

There are serious problems:

- **`lambda`** doesn't work right

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

- It's easy to accidentally depend on dynamic bindings
- It's easy to accidentally override a dynamic binding

The last two are unacceptable for large systems

$\Rightarrow$ make your language statically scoped

A Little Dynamic Scope Goes a Long Way

Sometimes, the programmer really needs dynamic scope:

```
(define (notify user msg)
  ; Should go to the current output stream,
  ; whatever that is for the current process:
  (printf "Msg from ~a: ~a\n" user msg))
```

Static scope should be the implicit default, but supporting explicit dynamic scope is a good idea:

- In Common LISP, variables can be designated as dynamic
- In Racket, a special form can be used to define and set dynamic bindings:

```
(define x (make-parameter 0))
(define (f y)
  (+ y (x)))
(+ (f 1) (parameterize ([x 3])
           (f 2)))
```

Part III

Recursion

Factorial

```
(local [(define fac
          (lambda (n)
            (if (zero? n)
                1
                (* n (fac (- n 1))))))]
  (fac 10))
```

`local` binds both in the body expression and in the binding expression

Factorial

```
(let ([fac
        (lambda (n)
          (if (zero? n)
              1
              (* n (fac (- n 1)))))])
  (fac 10))
```

Doesn't work: **`let`** is like **`with`**

Still, at the point that we call **`fac`**, obviously we have a binding for **`fac`**...

... so pass it as an argument!

Factorial

```
(let ([facX
        (lambda (facX n)
          (if (zero? n)
              1
              (* n (facX facX (- n 1)))))])
  (facX facX 10))
```

Wrap this to get `fac` back...

Factorial

```
(let ([fac
       (lambda (n)
         (let ([facX
                (lambda (facX n)
                  (if (zero? n)
                      1
                      (* n (facX facX (- n 1)))))])
           (facX facX n)))])
  (fac 10))
```

Try this in the **HtDP Intermediate with Lambda** language, click **Step**

But the language we implement has only single-argument functions...

From Multi-Argument to Single-Argument

```
(define f
  (lambda (x y z)
    (list z y x)))

(f 1 2 3)
```

$\Longrightarrow$

```
(define f
  (lambda (x)
    (lambda (y)
      (lambda (z)
        (list z y x)))))

(((f 1) 2) 3)
```

Factorial

```
(let ([fac
       (lambda (n)
         (let ([facX
                (lambda (facX)
                  (lambda (n)
                   (if (zero? n)
                       1
                       (* n ((facX facX) (- n 1))))))])
           ((facX facX) n)))])
  (fac 10))
```

Simplify: `(lambda (n) (let ([f ...]) ((f f) n)))`

⇒ `(let ([f ...]) (f f))`...

Factorial

```
(let ([fac
       (let ([facX
              (lambda (facX)
                (lambda (n)
                  (if (zero? n)
                      1
                      (* n ((facX facX) (- n 1))))))])
         (facX facX))])
  (fac 10))
```

Factorial

```
(let ([fac
       (let ([facX
              (lambda (facX)
                ; Almost looks like original fac:
                (lambda (n)
                  (if (zero? n)
                      1
                      (* n ((facX facX) (- n 1))))))])
         (facX facX))])
  (fac 10))
```

More like original: introduce a local binding for `(facX facX)`...

Factorial

```
(let ([fac
       (let ([facX
              (lambda (facX)
                (let ([fac (facX facX)])
                  ; Exactly like original fac:
                  (lambda (n)
                    (if (zero? n)
                        1
                        (* n (fac (- n 1)))))))])
         (facX facX))])
  (fac 10))
```

Oops! — this is an infinite loop

We used to evaluate `(facX facX)` only when `n` is non-zero

Delay `(facX facX)` ...

Factorial

```
(let ([fac
       (let ([facX
              (lambda (facX)
                (let ([fac (lambda (x)
                             ((facX facX) x))])
                  ; Exactly like original fac:
                  (lambda (n)
                    (if (zero? n)
                        1
                        (* n (fac (- n 1)))))))])
         (facX facX))])
  (fac 10))
```

Now, what about **`fib`**, **`sum`**, etc.?

Abstract over the **`fac`**-specific part...

Make-Recursive and Factorial

```
(define (mk-rec body-proc)
  (let ([fX
         (lambda (fX)
           (let ([f (lambda (x)
                      ((fX fX) x))])
             (body-proc f)))])
    (fX fX)))

(let ([fac (mk-rec
             (lambda (fac)
               ; Exactly like original fac:
               (lambda (n)
                 (if (zero? n)
                     1
                     (* n (fac (- n 1)))))))])
  (fac 10))
```

Fibonnaci

```
(let ([fib
       (mk-rec
        (lambda (fib)
          ; Usual fib:
          (lambda (n)
            (if (or (= n 0) (= n 1))
                1
                (+ (fib (- n 1))
                   (fib (- n 2)))))))])
  (fib 5))
```

Sum

```
(let ([sum
       (mk-rec
        (lambda (sum)
          ; Usual sum:
          (lambda (l)
            (if (empty? l)
                0
                (+ (first l)
                   (sum (rest l)))))))])
  (sum '(1 2 3 4)))
```

Implementing Recursion

```
{rec {fac {fun {n}
               {ifzero n
                       1
                       {* n
                          {fac {- n 1}}}}}}
     {fac 10}}
```

could be parsed the same as

```
{with {fac
       {mk-rec
        {fun {fac}
             {fun {n}
                  {ifzero n
                          1
                          {* n
                             {fac {- n 1}}}}}}}}
   {fac 10}}
```

Implementing Recursion

```
{rec {<id>_1 <FAE>_1}
      <FAE>_2}
```

could be parsed the same as

```
{with {<id>_1 {mk-rec {fun {<id>_1} <FAE>_1}}}
  <FAE>_2}
```

which is really

```
{{fun {<id>_1} <FAE>_2}
 {mk-rec {fun {<id>_1} <FAE>_1}}}
```

which, writing out *mk-rec*, is really

```
{{fun {<id>_1} <FAE>_2}
 {{fun {body-proc}
       {with {fX {fun {fX}
                      {with {f {fun {x}
                                    {{fX fX} x}}}
                        {body-proc f}}}}}
         {fX fX}}}
  {fun {<id>_1} <FAE>_1}}}
```