Part I
Lexical Addresses and Compilation
Identifier Address

Suppose that

\{ \text{fun} \ \{x\} \ \{+ \ x \ y\} \}\n
appears in a program...

If the body is eventually evaluated:

\{+ \ x \ y\}

where will \textit{x} be in the substitution?

\textbf{Answer:} always at the beginning:

\texttt{x = \ldots \ldots}
Identifier Address

Suppose that

\{\textit{with} \ \{y \ 1\} \ \{+ \ x \ y\}\}

appears in a program...

If the body is eventually evaluated:

\{+ \ x \ y\}

\textit{where} will \textit{y} be in the substitution?

\textbf{Answer:} always at the beginning:

\textit{y} = 1 \ \ldots
Identifier Address

Suppose that

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

appears in a program...

If the body is eventually evaluated:

{ + x y}

where will \textit{y} be in the substitution?

\textbf{Answer:} always second:

\begin{align*}
x &= \ldots \quad y &= 1 \quad \ldots
\end{align*}
Identifier Address

Suppose that

\[
\{\text{with } \{y \ 1\} \ \\
\ {\{\text{fun } \{x\} \ \{- \{+ \ x \ y\} \ 17\}\} \ 88\}\}
\]

appears in a program...

If the body is eventually evaluated:

\[
\{+ \ x \ y\}
\]

where will \( x \) and \( y \) be in the substitution?

**Answer:** always first and second:

\[
x = \ldots \quad y = 1 \quad \ldots
\]
Identifier Address

Suppose that

\[
\{\text{with } \{y \ 1\} \\
\{\{\text{fun } \{w\} \ \{\text{with } \{z \ 9\} \\
\{\text{fun } \{x\} \ \{+ \ x \ y\}\}\}\}\}\}
\]

appears in a program...

If the body is eventually evaluated:

\[
\{+ \ x \ y\}
\]

where will \(x\) and \(y\) be in the substitution?

**Answer:** always first and fourth:

\[
x = \ldots \\
z = 9 \\
w = \ldots \\
y = 1 \\
\ldots
\]
Suppose that

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

appears in a program...

If the body is eventually evaluated:

```
{+ x y}
```

where will \(x\) and \(y\) be in the substitution?

**Answer:** always first and fourth:
Lexical Scope

Our language is *lexically scoped*:

- For any expression, we can tell which identifiers will have substitutions at run time
- The order of the substitutions is also predictable

(The value for each substitution is not necessarily predictable)
Compiling FAE

A *compiler* can transform an **FAE** expression to an expression without identifiers – only lexical addresses

; 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

(\text{compile } 1 \ldots) \Rightarrow 1

(\text{compile } \{+ 1 2\} \ldots) \Rightarrow \{+ 1 2\}

(\text{compile } x \ldots) \Rightarrow \text{compile: free identifier}

(\text{compile } \{\text{fun } \{x\} x\} \ldots) \Rightarrow \{\text{fun } \{\text{at 0}\}\}

(\text{compile } \{\text{fun } \{y\} \{\text{fun } \{x\} \{+ x y\}\}\} \ldots)

\Rightarrow \{\text{fun } \{\text{fun } \{+ \{\text{at 0}\} \{\text{at 1}\}\}\}\}\}

(\text{compile } \{\{\text{fun } \{x\} x\} 10\} \ldots)

\Rightarrow \{\{\text{fun } \{\text{at 0}\}\} 10\}
; 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))])))
Compile-Time Substitution

Mimics run-time substitutions, but without values:

\[
\text{(define-type CSubs}
\begin{array}{l}
\text{[mtCSub]}
\text{[aCSub (name symbol?)}
\text{(rest CSubs?)]]}
\end{array}
\]\n
; locate : symbol CSubs -> number
\[
\text{(define (locate name cs)}
\begin{array}{l}
\text{(type-case CSubs cs}
\begin{array}{l}
\text{[mtCSub ()}
\text{\quad (error 'compile "free identifier")]}
\text{[aCSub (sub-name rest)\quad}
\text{(if (symbol=? name sub-name)
\text{\quad 0
\text{\quad (+ 1 (locate name rest))]))])}
\end{array}
\end{array}
\)
CFAE Values

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

```scheme
(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)))))))
```
CFAE Versus FAE Interpretation

On my machine,

\[(\text{cinterp}\ \text{empty})\]

is 30% faster than

\[(\text{interp}\ \text{(mtSub)})\]

Note: using built-in \texttt{list-ref} simulates machine array indexing, but don’t take the numbers too seriously
Part II
Dynamic Scope
Recursion

What if we want to write a recursive function?

```plaintext
{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

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

⇒
{f 0}
```

Lexical scope:

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

⇒
```

Dynamic scope:

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

⇒
```

32-35
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:

```plaintext
{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

  ```scheme
  (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

⇒ 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 PLT and other Schemes, special forms 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)))
```