Identifier Address

Suppose that

\{with \{x 88\} \{ + 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:

\(x = 88\) \ldots
Identifier Address

Suppose that

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

appears in a program; the body is eventually evaluated:

\{+ \ x \ y\}

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}
 {with {x 2} {+ x y}}}
```

appears in a program; the body is eventually evaluated:

```
{+ x y}
```

where will `y` be in the substitution?

**Answer:** always second:

```
x = 2  y = 1  ...
```
Identifier Address

Suppose that

\[
\begin{align*}
\{ \text{with } \{ y \ 1 \} \\
\{ \text{with } \{ x \ 88 \} \ - \ \{ + \ x \ y \} \ 17 \} \}
\end{align*}
\]

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:

\[
x = 88 \quad y = 1 \quad \ldots
\]
Identifier Address

Suppose that

\[
\{\text{with } \{y \ 1\}\}
\{\text{with } \{w \ 10\}\}
\{\text{with } \{z \ 9\}\}
\{\text{with } \{x \ 0\}\ \{+ \ x \ y\}\}\}\}
\]

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:

\[
x = 0 \quad z = 9 \quad w = 10 \quad y = 1 \quad \ldots
\]
Identifier Address

Suppose that

\[
\text{\{with \ \{y \ \{with \ \{r \ 9\} \ \{- \ r \ 8\}\}\}\}}
\text{\{with \ \{w \ 10\} \ \{with \ \{z \ \{with \ \{q \ 9\} \ q\}\}\}
\text{\{with \ \{x \ 0\} \ \{+ \ x \ y\}\}\}\}}
\]

appears in a program; the body is eventually evaluated:

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

where will \text{x} and \text{y} be in the substitution?

\textbf{Answer:} always first and fourth:

\[
\text{x = 0 \ z = 9 \ w = 10 \ y = 1 \ \ldots}
\]
Lexical Scope

Our language is \textit{lexically scoped}:

\begin{itemize}
  \item For any expression, we can tell which identifiers will have substitutions at run time
  \item The order of the substitutions is also predictable
\end{itemize}
Compiling F1WAE

A **compiler** can transform an **FW1AE** expression to an expression without identifiers — only lexical addresses

; compile : F1WAE ... -> CF1WAE

```
(define-type F1WAE
 [num (n number?)])
[add (lhs F1WAE?)
 (rhs F1WAE?)]
[sub (lhs F1WAE?)
 (rhs F1WAE?)]
[with (name symbol?)
 (named-exp F1WAE?)
 (body F1WAE?)]
[id (name symbol?)]
[app (fun-name symbol?)
 (arg-exp F1WAE?)]

(define-type CF1WAE
 [cnum (n number?)])
[cadd (lhs CF1WAE?)
 (rhs CF1WAE?)]
[csub (lhs CF1WAE?)
 (rhs CF1WAE?)]
[cwith (named-exp CF1WAE?)
 (body CF1WAE?)]
[cat (pos number?)]
[capp (fun-name symbol?)
 (arg-exp CF1WAE?)]
```
Compile Examples

(compile 1 ...) ⇒ 1

(compile [+ 1 2] ...) ⇒ [+ 1 2]

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

(compile {with {x 8} x} ...) ⇒ {with 8 {at 0}}

(compile {with {y 1} {with {x 2} [+ x y]}} ...) => {with 1 {with 2 [+ {at 0} {at 1}]}{at 1}}}

(compile {deffun {f x} x} ...) → {deffun f {at 0}}
Implementing the Compiler

; compile : F1WAE CSub -> CF1WAE
(define (compile a-wae cs)
  (type-case F1WAE a-wae
    [num (n) (cnum n)]
    [add (l r) (cadd (compile l cs) (compile r cs))]
    [sub (l r) (csub (compile l cs) (compile r cs))]
    [with (named named-expr body-expr)
      (cwith (compile named-expr cs) (compile body-expr
        (aCSub named cs)))]
    [id (name) (cat (locate name cs))]
    [app (fun-name arg-expr)
      (capp fun-name (compile arg-expr cs))])))
Compile-Time Substitution

Mimics run-time substitutions, but without values:

```scheme
(define-type CSub
    [mtCSub]
    [aCSub (name symbol?)
        (rest CSub?)])

; locate : symbol CSub -> number
(define (locate name cs)
    (type-case CSub cs
        [mtCSub ()
            (error 'compile "free identifier")]
        [aCSub (sub-name rest)
            (if (symbol=? name sub-name)
                0
                (+ 1 (locate name rest)))]))
```
CFI WAE Interpreter

Almost the same as **F1WAE interp**:

```scheme
; cinterp : CFIWAE list-of-num -> num
(define (cinterp a-cwae s)
  (type-case CFIWAE a-cwae
    [cnum (n) n]
    [cadd (l r) (+ (cinterp l s) (cinterp r s))]
    [csub (l r) (- (cinterp l s) (cinterp r s))]
    [cwith (named-expr body-expr)
      (cinterp body-expr
cfundefs
        (cons (cinterp named-expr cfundefs s)
s))]
    [cat (pos) (list-ref s pos)]
    [capp (fun-name arg)
      (local [(define fun (lookup-cfundef fun-name cfundefs))
        (define arg-val (cinterp arg cfundefs s))]
        (cinterp (cfundef-body fun)
cfundefs
          (cons arg-val empty))))]
```
CFWAE Versus FIWAE Interpretation

On my machine,

```
(cinterp
  {with {x 1} {with {y 2} {with {z 3} {{+ {+ x x} {+ x x}}}}}}
empty)
```

takes about half the time of

```
(interp
  {with {x 1} {with {y 2} {with {z 3} {{+ {+ x x} {+ x x}}}}}}
(mtSub))
```

Note: using built-in `list-ref` simulates machine array indexing, but don’t take the numbers too seriously