Lexical Addresses

As we saw in the last lecture, the expression

```
let x = 1  y = 2
  in let f = proc (x) +(x, y)
       in (f x)
```

might be compiled to

```
let _ = 1 _ = 2
  in let _ = proc (_) +(<0,0>, <1,1>)
      in (<0,0> <1,0>)
```

\(<n, m>\) means: \(n\) frames up in the environment, at position \(m\)

How can we compute \(<n, m>\) for every bound variable without running the code?
Computing Lexical Addresses

• What creates a new frame?

  \textbf{let, letrec, and (application of) proc}

• So, to compute the \( n \) in \(<n, m>\), count the number of enclosing \textbf{let}, \textbf{letrec}, and \textbf{proc} keywords between the bound variable and its binding

• The \( m \) in \(<n, m>\) is simply the variable's position in its binding set
Computing Lexical Addresses

Visualize as *countours* that separate environment extension from the expressions that use it

\[
\text{proc } (x) + (x, 7)
\]

- Count contour crossings to get \( n + 1 \)
- Cross 1 contour from bound \( x \) to binding \( x \), so first part of address is 0
- Full address is \( <0, 0> \)
Computing Lexical Addresses

Visualize as *countours* that separate environment extension from the expressions that use it

\[
\text{proc } (y) \quad \text{proc } (x, z) \quad + (x, -(y, z))
\]

- Bound \( x \): \( <0, 0> \)
- Bound \( y \): \( <1, 0> \)
- Bound \( z \): \( <0, 1> \)
Computing Lexical Addresses

Visualize as *countours* that separate environment extension from the expressions that use it

\[
\text{proc } (y) \quad \text{proc } (x, z) \quad +\left(x, -(y, z)\right)
\]

In general:

\[
\text{proc } (\langle\text{id}\rangle_1, ..., \langle\text{id}\rangle_n) \quad \langle\text{expr}\rangle
\]
Computing Lexical Addresses

Visualize as *countours* that separate environment extension from the expressions that use it

```plaintext
let x = 5
    in x
```

In general:

```plaintext
let <id>_1 = <expr>_1
    ...
    ... = ...
    <id>_n = <expr>_n
    in <expr>
```
Computing Lexical Addresses

Visualize as *countours* that separate environment extension from the expressions that use it

```plaintext
let x = 5
    in x
```

- Bound $x$: $<0, 0>$
Computing Lexical Addresses

Visualize as *countours* that separate environment extension from the expressions that use it

```plaintext
let x = 5
  y = 7
in let x = x
  in +(x, y)
```
Computing Lexical Addresses

Visualize as *countours* that separate environment extension from the expressions that use it.

```
let x = 5
y = 7
in let x = x
    in +(x, y)
```

- Bound `x`: <0, 0>
- Bound `x`: <0, 0>
- Bound `y`: <1, 1>
Computing Lexical Addresses

Visualize as **countours** that separate environment extension from the expressions that use it.

\[
\text{letrec } f = \text{proc } (x) \cdot (+ (x, (g \ 7))) \\
\quad g = \text{proc } (z) \cdot (- (z, 2)) \\
\quad \text{in } (f \ 10)
\]

In general:

\[
\text{letrec } <id>_1 = <expr>_1 \\
\quad \ldots = \ldots \\
\quad <id>_n = <expr>_n \\
\quad \text{in } <expr>
\]
Computing Lexical Addresses

Visualize as *countours* that separate environment extension from the expressions that use it.

\[
\text{letrec } f = \begin{array}{c}
\text{proc } (x) \\
(x, (g\ 7)) \\
\end{array}
\]

\[
\text{g = proc } (z) \\
(z, 2) \\
\end{array}
\]

\[
\text{in } (f\ 10)
\]

- Bound *x*: <0, 0>
- Bound *g*: <1, 1>
- Bound *z*: <0, 0>
- Bound *f*: <0, 0>
Lexical Addresses are Static

- The contour approach to computing lexical addresses works because they are *static*
- That's why we can pre-compute them in a compiler
Source Language for Compilation

<expr> ::= <num>
 ::= <id>
 ::= <prim> ( { <expr> }* )
 ::= let { <id> = <expr> }* in <expr>
 ::= proc ( { <id> }* ) <expr>
 ::= (<expr> <expr>* )

concrete
Source Language for Compilation

<expr> ::= (lit-exp <num>)
 ::= (var-exp <symbol>)
 ::= (primapp-exp <prim> (list <expr>*))
 ::= (let-exp (list <symbol>*) (list <expr>*) <expr>)
 ::= (proc-exp (list <symbol>*) <expr>)
 ::= (app-exp <expr> (list <expr>*))

abstract
Target Language for Compilation

\[
\begin{align*}
\text{<cexpr> ::=} & \ (\text{lit-cexp <num>}) \\
\text{::=} & \ (\text{var-cexp <num> <num>}) \\
\text{::=} & \ (\text{primapp-cexp <prim> (list <cexpr>*))} \\
\text{::=} & \ (\text{let-cexp (list <cexpr>* <cexpr>}) \\
\text{::=} & \ (\text{proc-cexp <cexpr>}) \\
\text{::=} & \ (\text{app-cexp <cexpr> (list <cexpr>*))}
\end{align*}
\]

\text{abstract}

(no use for concrete)

For implementation: declare a \textit{cexpression} datatype with \textit{define-datatype}
Compilation Function

\[ \text{compile-expression} : \text{expr} \rightarrow \text{cexpr} \]

- Mostly trivial: create a \(<\text{cexpr}>\) corresponding to the input \(<\text{expr}>\)
- Interesting case: \textit{var-exp}
  - Use an environment, almost like evaluation
  - Key difference #1: instead of \textit{apply-env}, we need \textit{lexical-address-in-env}
  - Key difference #2: no closures; instead, compile a \textit{proc} body immediately when we encounter the \textit{proc}
Evaluation Function for the Target Language

• `eval-cexpression` is similar to `eval-expression`, except:
  ○ The names in the environment do not matter
  ○ Use `apply-env-to-lexical-address` instead of `apply-env`
Implementation

(implement in DrScheme)