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

<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?

• What creates a new frame?

let, letrec, and (application of) proc

- So, to compute the n in <n, m>, count the number of enclosing let, letrec, and proc keywords between the bound variable and its binding
- The **m** in <**n**, **m**> is simply the variable's position in its binding set

- 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>

- Bound **x**: <0, 0>
- Bound **y**: <1, 0>
- Bound **z**: <0, 1>

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

In general:

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

In general:

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

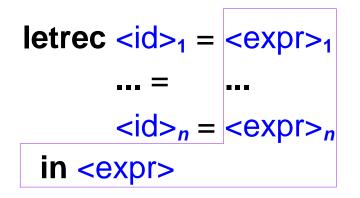
$$\begin{array}{c} \text{let } \mathbf{x} = 5 \\ \text{in } \mathbf{x} \end{array}$$

• Bound **x**: <0, 0>

- Bound **x**: <0, 0>
- Bound **x**: <0, 0>
- Bound **y**: <1, 1>

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

In general:



- 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

- <cexpr> ::= (lit-cexp <num>)
 - ::= (var-cexp <num> (num>)
 - ::= (primapp-cexp <prim> (list <cexpr>*))
 - ::= (let-cexp (list <cexpr>*) <cexpr>)
 - ::= (proc-cexp <cexpr>)
 - ::= (app-cexp <cexpr> (list <cexpr>*))

abstract

(no use for concrete)

For implementation: declare a cexpression datatype with define-datatype

Compilation Function

compile-expression : expr -> cexpr

- Mostly trival: create a <cexpr> corresponding to the input <expr>
- Interesting case: var-exp
 - Use an environment, almost like evaluation
 - Key difference #1: instead of apply-env, we need lexical-address-in-env
 - Key difference #2: no closures; instead, compile a proc body immediately when we encounter the 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)