What is a Binding?

- A *binding* is an association between an identifier (or variable) and a value.
- We say the identifier is “bound to” the value.

- Examples:
  - X is bound to 5
  - foo is bound to “bar”
  - null is bound to OpenList.nil
What is an Environment?

• An environment is a set of bindings.

Why is this important?

• Environments change in going from one part of an expression to another.

• For the most part, such changes can be accomplished non-destructively.

• Old environments are not destroyed, just modified by layering on new bindings.
How is an environment represented?

- There are multiple ways, but in the Rockit interpreter, we use an association list.

- `'((X 5) (Y 10) (Z 15) (X 20))`
  - X is bound to 5
  - Y is bound to 10
  - Z is bound to 15

  The remaining X has no effect, as association lists are searched front to back.

Changing Environments

- These forms all have a temporary effect on the environment:
  - lambda
  - let
  - let*
  - letrec
Example 1: let

Say the ambient environment is
‘((X 5) (Y 10) (Z 15) (X 20))

Consider the expression
(let ((Y 30)) (+ X Y Z))
Example 2: nested lets

Consider the expression

(let ((Y 30)) (let ((Z 0)) (+ X Y Z)))

```
(((X 5) (Y 10) (Z 15))

outer environment
```

Example 2: nested lets

Consider the expression

(let ((Y 30)) (let ((Z 0)) (+ X Y Z)))

```
(((Y 30)(X 5) (Y 10) (Z 15))

mezzanine body environment
```

```
((X 5) (Y 10) (Z 15))

outer environment
```
Example 2: nested lets

Consider the expression

\[
\text{(let ((Y 30)) (let ((Z 0)) (+ X Y Z)))}
\]

\[
\text{((Y 30) (X 5) (Y 10) (Z 15))}
\]

\[
\text{((X 5) (Y 10) (Z 15))}
\]

outer environment

mezzanine body environment

innermost body environment

Example 3: Variables in right-hand sides of equations

- The variable bindings defined by let expressions can be viewed as determined from “equations”:

\[
\text{(let ((Y 30)) …) is like Y = 30}
\]

The right-hand sides can contain variables

\[
\text{(let ((Y (* Z Z)) …) is like Y = (* Z Z)}
\]
Example 3: Variables in right-hand sides of equations

- This raises the question of “In what environment are the right-hand sides evaluated?”

- The answer is: in the environment containing the let expression.

Example 3: nested lets

Consider the expression

\[
\text{(let ((Y (+ X 2))) (let ((Z (+ X Y))) Z))}
\]

\[
'(\text{(X 5) (Y 10) (Z 15)})
\]

outer environment
Example 3: nested lets

Consider the expression

(let ((Y (+ X 2))) (let ((Z (+ X Y))) Z))

Expand it:

(let ((Y (+ X 2))) (let ((Z (+ X Y))) Z))

outer environment

= (((Y 7) (Z 12)) (Y 7) (X 5) (Y 10) (Z 15))

mezzanine body environment

= (((Z 12) (Y 7) (X 5) (Y 10) (Z 15))

innermost body environment

= (((Y 7) (X 5) (Y 10) (Z 15))

mezzanine body environment

= (((X 5) (Y 10) (Z 15))

outer environment

= (((X 5) (Y 10) (Z 15))

outer environment

= (((X 5) (Y 10) (Z 15))

outer environment
Example 4: “Redefining” lets

\[
\text{(let \((X (* X X))) X)}
\]

\[
\text{\'((X 25) (X 5))}
\]

\[
\text{\'((X 5)}
\]

Layering

• When interpreting a let expression, it is not necessary to change the outer environment.

• Instead, just lay on new bindings for evaluating the body of the let.

• When evaluation is complete, the original environment will still be intact.
Let* vs. Let

- let* is like a cascade of nested lets:
- \((\text{let* } ((X \ 5) \ (Y \ (* \ X \ X)) \ (X \ (+ \ X \ Y)) \ \ldots))\)
  is equivalent to
  \((\text{let } ((X \ 5))\)
  \(\left(\text{let } ((Y \ (* \ X \ X)))\right)\)
  \(\left(\text{let } ((X \ (+ \ X \ Y)))\right)\)
  \(\ldots\)
  \(\right)\)
  \(\right)\)
  \(\right)\)

Letrec

- In letrec, the environment for the RHS is the same as the inner environment.
- \((\text{letrec } ((f \ (\lambda(x)\)
  \(\quad \text{(if } (< \ x \ 2) \ 1 \ (* \ x \ (f \ (- \ x \ 1)))))))\)
  \(\quad (f \ 5)\)
  \)}
works, but will not if letrec is replaced with let. Why?
letrec vs. let

> (letrec (((f (lambda(x) (if (< x 2) 1 (* x (f (- x 1)))))) (f 10))
3628800

> (let (((f (lambda(x) (if (< x 2) 1 (* x (f (- x 1))))))) (f 10))
  . f: undefined;
cannot reference an identifier before its definition

Using define

• Will this work?
> (define f (lambda(x) (if (< x 2) 1 (* x (f (- x 1))))))

> (f 10)
The base environment

- *define*, at the top level, adds bindings to the base environment.

- The base environment is **used if no binding exists otherwise**.

```scheme
> (define f (lambda(x) (if (< x 2) 1 (* x (f (- x 1)))))
> (f 10)
3628800
```

In Rockit Interpreter

- In the variable lookup part of eval, have it check the base environment after checking the environment passed to eval.
public static Object eval(Object ob, OpenList env)
{
    if( ob instanceof OpenList )
    {
        // An OpenList expression is some kind of form
        return evalForm((OpenList)ob, env);
    }

    if( ob instanceof String )
    {
        // A String is a variable
        OpenList found = env.assoc(ob);
        if( found.nonEmpty() )
        {
            return found.second();
        }
    }

    // Checking the base environment after the environment arg enables
    // recursion in a natural way.
    found = baseEnvironment.assoc(ob);
    if( found.nonEmpty() )
    {
        return found.second();
    }
}

Lambda Expressions

• Lambda expressions translate into function objects (usually called “closures”) that carry their environment with them.

• Sometimes this matters, sometimes not (when?)

• The base environment is still checked if a binding is not found.
Bindings for Lambda Expressions

\[(\text{lambda} \ (X) \ (+ \ X \ Y))\]

\[('((X \ 6) \ (Y \ 5)) \ ('((X \ ???) \ (X \ 6) \ (Y \ 5))'\]

ambient environment (for example)  body environment

The body environment is only known when the function is applied.

Bindings for Lambda Expressions

\[(\text{lambda} \ (X) \ (+ \ X \ Y)) \ 99\]

\[('((X \ 6) \ (Y \ 5)) \ ('((X \ 99) \ (X \ 6) \ (Y \ 5))'\]

ambient environment  body environment

The body environment is only known when the function is applied.
How to Bind Lambdas

If there is a keyword `lambda` immediately inside the opening left paren, the expression represents a function. Otherwise it is an application.

Start from the outermost application that can be evaluated and work inward, adding binding layers as you go.

**Bindings can also be represented by replacing free variables with their values.**

Constructing/Deconstructing Examples

- `(lambda (X) …) function`
- `(((lambda (X) …) 4) application`
- If the above function returns a function, then `(((lambda (X) …) 4) 5)` is that function applied.

Example: function returning a function

- `(((lambda (X) (lambda(Y) (list X Y)) 4) 5)
  (lambda(Y) (list 4 Y)) 5)
  (list 4 5)`

  [This is the “Curried” version of the list function.]
Similarity is not Identity

\(((\text{lambda} (X) (\text{lambda}(Y) (\text{list} X Y)) 4)) 5\) before vs.
\((\text{lambda}(X) ((\text{lambda}(Y) (\text{list} X Y)) 4)) 5\) now

In the second case, the body of the outermost function is itself the \textbf{result} of an application:
\(((\text{lambda}(Y) (\text{list} 5 Y)) 4))\), but not a function itself.

Gives ‘(5 4), not ‘(4 5) as before.

Tree Representation

• When all else fails, expressions can be parsed as a \textbf{tree}.

\((\text{lambda} (X) \ldots)\) function

\(\lambda X\)

\(\ldots\)

\(\ldots\) application

Applies a function \(\ldots\) to \ldots
Tree Example

- (((lambda (X) (lambda(Y) (list X Y)) 4) 5)

5

((lambda (X) (lambda(Y) (list X Y)) 4)

application, or function?

Tree Example

- (((lambda (X) (lambda(Y) (list X Y)) 4) 5)

5

4

(l(lambda (X) (lambda(Y) (list X Y)))

application, or lambda exp?
Tree Example

• (((lambda (X) (lambda(Y) (list X Y)) 4) 5)

 application, or lambda exp?
Evaluating

- Application **annihilates** lambda

```
  \lambda X \ldots
  ___
```

Substitute … for all **free** occurrences of `X` in ___.

[This step is called **\(\beta\) reduction** in CS & Logic.]

---

Evaluating

```
  \lambda X

  5

  \lambda Y

  sub for X

  4

  \lambda Y

  \lambda

  5

  (list X Y)

  (list 4 Y)

  \beta
```
Evaluating

\[ \lambda Y \ (\text{list } 4 \ Y) \]

- sub for Y
- \( \beta \) (list 4 5)

Example

- \(((\lambda X) \ ((\lambda Y) \ (\text{list } X \ Y)) \ 4)) \ 5)\n
- Construct tree, then evaluate.
Example

- $((\lambda(X) \ ((\lambda(Y) \ (\text{list} \ X \ Y)) \ 4)) \ 5)$
  Application or function?
Example

- \(((\text{lambda}(X) ((\text{lambda}(Y) (\text{list} X Y)) 4)) 5)\)

```
\lambda X

(((\text{lambda}(Y) (\text{list} X Y)) 4)
```

Example

- \(((\text{lambda}(X) ((\text{lambda}(Y) (\text{list} X Y)) 4)) 5)\)

```
\lambda X

(((\text{lambda}(Y) (\text{list} X Y)) 4)
```

```
5

4

((\text{lambda}(Y) (\text{list} X Y))
```
Example

- \(((\text{lambda}(X) ((\text{lambda}(Y) (\text{list} X Y)) 4)) 5)\)

Two Trees Compared
Example

• \(((\lambda(X) \ ((\lambda(Y) \ (\text{list } X \ Y)) \ 4)) \ 5)\)

\[
\begin{align*}
\text{list } X \ Y & \quad \lambda X \ 5 \quad \text{sub for } X \\
\lambda Y & \ 4 \\
\end{align*}
\]

Example

• \(((\lambda(X) \ ((\lambda(Y) \ (\text{list } X \ Y)) \ 4)) \ 5)\)

\[
\begin{align*}
\text{list } 5 \ Y & \quad \lambda Y \ 4 \\
\text{list } 5 \ Y & \quad \beta \\
\end{align*}
\]

vs.

\[
\begin{align*}
\text{list } 4 \ 5 \\
\text{for other tree}
\end{align*}
\]
Static vs. Dynamic Binding

- Mathematical functions demand static binding

(define y 99)
(define f (lambda(x) (+ x y)))
(let ((y 1000)) (f 1))

y in (+ x y) should be bound to 99 (static binding)

If it were dynamic binding, y would be bound to 1000 in (+ x y).

Static Binding Example 1

(let ((X 4))
  (lambda(Y) (list X Y)) 5)
)

The application applies a function
  (lambda(Y) (list X Y)),
  with the static binding (X 4).

This function applied to 5 gives ‘(4 5).
Static Binding Example 2

(let ((X 4))
  (let ((f (lambda(Y) (list X Y))))
    (f 5))
)

Same deal: The application applies a function
(lambda(Y) (list X Y)),
with the static binding (X 4).

This function applied to 5 gives ‘(4 5).

Static Binding Example 3

(let ((X 4))
  (let ((f (lambda(Y) (list X Y))))
    (let ((X 99))
      (f 5))
  )
)

Still same deal: The application applies a function
(lambda(Y) (list X Y)),
with the static binding (X 4).

This function applied to 5 gives ‘(4 5).
Dynamic Binding Example 1

\[
\text{(let ((X 4))}
\text{  (lambda(Y) (list X Y)) 5)}
\]

The application applies a function \textit{expression} \text{(lambda(Y) (list X Y))}.

X is bound to 4 in the environment of the application, so that is used.

This function applied to 5 gives ‘(4 5).

Dynamic Binding Example 2

\[
\text{(let ((X 4))}
\text{  (let ((f (lambda(Y) (list X Y))))}
\text{    (f 5))}
\]

Same deal: The application applies a function \textit{expression} \text{(lambda(Y) (list X Y))}

X is bound to 4 in the environment of the application, so that is used.

This function applied to 5 gives ‘(4 5).
Dynamic Binding Example 3

(let ((X 4))
  (let ((f (lambda(Y) (list X Y))))
    (let ((X 99))
      (f 5)
    )
  )
)

Different: The application is a function expression.
(lambda(Y) (list X Y)),
The application environment has X bound to 99.
The result is ‘(99 5), not ‘(4 5) the mathematical result.

Quasi-Static bindings

• In Racket, bindings in the base environment are statically bound to locations (variables) rather than values.

> (define x 99)
> (define g (lambda(y) (+ x y)))
> (g 1)
100

> (set! x 100) ; destructively modifies the x value
> (g 1)
101 ; The location hasn’t changed,
; but the value has.
Let as Lambda

- Lambda is more general than Let

- \((\text{let } ((V_1 E_1) (V_2 E_2) \ldots (V_n E_n)) E_0)\)

  \[=\]

  \((\lambda (V_1 V_2 \ldots V_n) E_0)\)

  \((E_1 E_2 \ldots E_n)\)

However, the static binding issue does not arise with let.

Double Example

- \((\text{define } double (\lambda (f) (\lambda (x) (f (f x)))))\)

- Draw the tree

- Evaluate \((\text{(double square) 5})\)

- Evaluate \((\text{(double double) square) 5})\)
(double double)

> (double double)
#<procedure>

> (((double double) square) 5)
152587890625

> (square (square (square (square 5)))))
152587890625

How Big?

(((double (double double)) square) 5)
How Big?

shot from my screen at 7-point font:

```
> (/ (log (((double (double double)) square) 5)) (log 10))
45,807 ; decimal digits
```

Composing Functions Using Functions

```
> (define (compose f g) (lambda (x) (f (g x))))
> (define (cube x) (* x x x))
> ((compose cube square) 5)
15625

> ((compose square cube) 5)
15625

> (define (add2 n) (+ 2 n))
> ((compose add2 square) 5)
27

> ((compose square add2) 5)
49
```