

Principles of Programming Languages Handout

First of all, we introduce some handy syntax: we make local definitions using `let`. For example, we let `x` be 2 and `y` be 5 before adding them:

```
(let ((x 2) (y 5))
      (+ x y))
```

This evaluates to 7. In fact, the `let`-notation is merely “syntactic sugar” for a `lambda` that is immediately applied:

```
((lambda (x y)
      (+ x y))
  2 5)
```

This is equivalent to the `let` above.

Note that `let` can be nested, and that the inner binding for `x` shadows the outer one:

```
(let ((x 2))
      (+
        (let ((x 5))
              x)
          x))
```

This evaluates to 7 again. We could equivalently have written

```
(let ((x 2))
      (+
        (let ((y 5))
              y)
          x))
```

This nested `let` is completely different from assigning to `x`, which we could write as follows:

```
(let ((x 2))
      (+
        (begin
              (set! x 5)
              x)
          x))
```

This evaluates to 10. The assignment *changes* the value of `x`, instead of introducing another variable also called `x`.

Now for a more subtle example. What does this evaluate to?

```
(let ((x 1))
      (let ((p (lambda (y) x)))
        (let ((x 2))
              (p 0))))
```

Note that there are two different bindings around. When the procedure `p` is applied to 0, which of these should be used for `x`?

In Scheme, the binding at the point of definition (which binds `x` to 1) is used. So the whole expression evaluates to 1.

Scheme (and Common Lisp) differ fundamentally from more archaic Lisps (such as the one in Emacs). These older Lisps still use dynamic binding. Apart from inserting the keyword `funcall` in the call of `p`, the above is also legal Emacs Lisp:

```
(let ((x 1))
  (let ((p (lambda (y) x)))
    (let ((x 2))
      (funcall p 0))))
```

But its meaning is different: it evaluates to 2. When the body of `p` is evaluated, the binding for `x` at the point of call (which binds it to 2) is used.

Similarly, in Scheme, the following expression evaluates to 1:

```
((let ((x 1))
  (lambda (y) x))
 0)
```

But in Emacs Lisp, the analogous expression

```
(funcall
  (let ((x 1))
    (lambda (y) x))
 0)
```

yields an error: there is no binding around for `x` when the call is evaluated.

One consequence of this is that in Scheme we can rename variables in their scope, just like α -conversion in the λ -calculus. For instance both of the `x` could be renamed:

```
(let ((a 1))
  (let ((p (lambda (y) a)))
    (let ((b 2))
      (p 0))))
```

On the other hand, the meaning of a procedure such as `p` cannot be just the expression

```
(lambda (y) x)
```

We need to keep track of what value is to be associated with `x`. If we do not, we get old-style Lispish behaviour. In the interpreter, *closures* are used to keep track of bindings for the free variables of procedure.

Note: you can evaluate Lisp expressions in Emacs by typing Control-x Control-e while the cursor is after the expression to be evaluated.