Lambda Expressions

We have often been defining procedures using the shorthand:

```
(define (square x)
    (* x x))
```

But recall that this is just shorhand for binding the variable `square` to the lambda expression `(* x x)`.

```
(define square
    (lambda (x)
       (* x x)
    )
)
```

It is often very useful to define procedures without naming them. These **anonymous procedures** can be passed as arguments, returned as arguments, bound to local variable names using let, etc. We will see further applications later when we cover higher-order procedures.

Lambda Expressions Examples

Establishing a procedure as the value of a local variable.

```
(let ((square-it (lambda (x) (* x x))))
   (list (square-it (+ 1 3))
         (square-it (* 2 5))
         (square-it 7)))       => (16 100 49)
```

`square-it` is defined only within the scope of the `let` statement.

Recall that procedures can have multiple arguments, and that we can even have **procedures as arguments** to procedures.

```
(let ((double-any (lambda (f x) (f x x))))
   (list (double-any + 25)
         (double-any cons 'a))) => (100 (a.a))
```

Dybvig §2.5 is a good reference to this material (available online). I **strongly** recommend that you read it. §4.2 may also be useful.

Lambda Expressions Examples (cont.)

The following examples are taken from Dybvig §2.5:

```
(let ((x 'a))
  (let ((f (lambda (y) (list x y))))
    (f 'b)))        returns     (a b)
```

Note that x is bound in the outer `let`. It is a *free variable* in the lambda expression. A variable that occurs free in a lambda expression should be bound by an enclosing lambda or let expression, unless the variable is (like the names of primitive procedures) bound at top level, as we discuss in the following section.

```
(let ((f (let ((x 'a))
           (lambda (y) (cons x y)))))
  (let ((x 'i-am-not-a))
    (f 'b)))  (a . b)
```

In both cases, the value of x within the procedure named f is a.

Interestingly, a let expression is just an application of a lambda expression to a set of argument expressions. I.e., the following two expressions are equivalent:

```
(let ((x 'a))
  (cons x x))

((lambda (x) (cons x x))
 'a)
```