define-syntax

define-syntax is a way to write a transformer that takes code written in one form and automatically transforms it into a different form that can be handled by the Scheme interpreter.

Other languages allow you to do something similar through "macros". For example, you can put at the top of a C program

```
#define INC(x) x++
```

Then if you have a variable bob in the program the line

```
INC(bob)
```

is replaced with

```
bob++
```

by the C-preprocessor

define-syntax has similar rewriting effects, but is much more powerful (of course).

It has the following form:

```
(define-syntax <keyword>
        (syntax-rules ()
                [pattern1  transformation1]
                [pattern2  transformation2]
                etc.))
```

Patterns can specify variables that can be used in the corresponding transformation.

Patterns usually start with _ which means that the use of the new syntax rule has to start with its keyword or name.
Patterns can include an ellipsis (3 dots) which mean that the previous symbol or subpattern can be repeated 0 or more times.

For example, here is a definition of the syntax of let, as it is built into the Scheme interpreter:

```
(define-syntax let
        (syntax-rules ( )
                [(_ ((x e) ...) b1 b2 ...)
                        ((lambda (x ...) b1 b2 ...) e ...)]))
```

Here is something a programmer might define. I want to take a sequence of numbers, find the factorial of each, and then collect them together into a single number using an operator. Altogether, I want (! 3 4 2 +) to represent (+ (Factorial 3) (Factorial 4) (Factorial 2)), or 32, while (! 3 4 2 *) is (* (Factorial 3) (Factorial 4) (Factorial 2)) or 288.

```
(define fact (lambda (n)
          (if (zero? n) 1 (* n (fact (sub1 n))))))

(define-syntax !
       (syntax-rules ()
             [(_ x y ... op) (op (fact x) (fact y) ...)]))
```

Here is another simple example. We want the form zero! to take a variable and sets it to 0, so

```
(define a 235)
(zero! a)
```

results in *a* having the value 0:

```
(define-syntax zero!
        (syntax-rules ()
                [(_ x) (set! x  0)]))
```

Note that this is very different from

```
(define set-to-zero (lambda (x) (set! x 0)))
```

If we say

```
(define b 23)
(set-to-zero b)
```

the value of *b* will remain 23.

Here is a more practical example:

```
(define-syntax when
        (syntax-rules ( )
                [(_ condition b c ...)
                        (if condition (begin b c ...) #f )]))
```

We might use this with

```
(when (< x 5) (println "That is small")  (set! x 10))
```

Finally, note that syntax rules don't evaluate their "arguments" the way functions do. With the *when* rule we just defined,

```
(define a 10)
(when (< a 5) (println "small") (set! a 10))
```

does not print "small". This will be important when we look at lazy evaluation and streams.