

More Elements of Scheme

Other Standard Functions

- `(length list)` returns the number of elements in a list. Example:

```
(length '(('munch 2 'apple 'pear "burp")))  
⇒ 5
```

```
(length '((x y (i j (a b c))))) ⇒ 1
```

- `(vector-length vector)` returns the number of elements in a vector. Example:

```
(vector-length '#(a b 32)) ⇒ 3
```

- `(string-length string)` returns the number of characters in a string. Example:

```
(string-length "Hello world!") ⇒ 12
```

Other Standard Functions (cont.)

- `(vector-ref vector n)` returns the element with index n . (N.B., as in C, the first element of a vector has index 0; the last has index $L - 1$, where L is the length of the vector.)

```
(vector-ref '#(3.6 1.2 -16. 4) 2) ⇒ -16.
```

- `(string-ref string n)` returns the character in string with index n . (N.B., again, the first element of a string has index 0; the last has index $L - 1$, where L is the length of the string.)

```
(string-ref "Scheme is fun!" 2) ⇒ #\h
```

- `(load "filename")` reads and evaluates the Scheme expressions contained in file *filename*. Example:

```
(load "my_scheme_lib.scm") ⇒ ?
```

Other Standard Functions (cont.)

- `(map function list_1 ... list_n)` applies a function of n arguments to n lists of equal length.

Example:

```
(map zero? '(0 0.3 -2 0)) ⇒ (#t #f #f #t)
```

```
(map abs '(0 0.3 -2 0)) ⇒ (0 0.3 2 0)
```

```
(map (lambda (x)
      (+ x 2)) '(0 0.3 -2 0))
```

```
⇒ (2 2.3 0 2)
```

```
(map (lambda (x y)
      (+ x y)) '(0 1 2) '(3 3 0))
```

```
⇒ (3 4 2)
```

```
(map (lambda (x y)
      (+ x y)) '(0 1 2) '(3 3))
```

```
⇒ *** ERROR -- Lists are not of equal
length.
```

Question

What does the following function do?

```
(define mystery
  (lambda (x)
    (map caddr x)))

(mystery '((a b c) (1 2 3) (x y z)))
```

Other Standard Functions (cont.)

- `apply` applies a function to a list such that the elements of the list are read as function arguments, e.g.,

```
(apply + '(1 2 3 4)) ⇒ 10
```

```
(apply max '(1 2 3 4)) ⇒ 4
```

```
(apply list '(1 2 3 4)) ⇒ (1 2 3 4)
```

Actually the syntax of `apply` is more generally of the form

```
(apply procedure object ... list),
```

the arguments of procedure are obtained by concatenating the objects and list elements in the order that they appear. Only one list is allowed. Thus,

```
(apply + 100 200 '(1 2 3 4)) ⇒ 310
```

Local Variables

On many occasions it is handy to define local variables to simplify the evaluation process, and create code that is easier to read. The keyword `let` facilitates this. Actually `(let varlist body)` creates a block of code that restricts the scope of all variable defined within its first argument. The syntax is explicitly,

```
(let ((var_1 value_1)
      (var_2 value_2)
      ...
      (var_n value_n))
  body)
```

Example: the Scheme expression

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

returns the number 17, without modifying the definitions of `x` and `y` elsewhere.

Functions that return functions

Consider the following Scheme definition

```
(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x))))))
```

where `f` and `g` are assumed to be functions that accept a single argument.

What happens if we type

```
(define mys (compose car cdr))
(mys '(a b c)) ?
```

Now consider the Scheme definition

```
(define f
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

What happens if we now evaluate

```
(define mystery (f 2))
```

and then

```
(mystery 9) ?
```

What about `((f 2) 9)`?

In a similar way, any function p of $n > 1$ arguments can be expressed as a function p' of $n - 1$ arguments if the value of the first argument is known. Symbollically,

$$((p' x_1) x_2 \dots x_n) = (p x_1 x_2 \dots x_n)$$

As a practical example, consider

```
(define expt-prime
  (lambda (x)
    (lambda (y)
      (expt x y))))
```

Then the following Scheme expressions are equivalent

$$((\text{expt-prime } 2) 10) = (\text{expt } 2 10)$$

Repeating this transformation $n - 1$ times yields a procedure p''

$$(\dots ((p'' x_1) x_2) \dots x_n) = (p x_1 x_2 \dots x_n).$$

This transformation is know as *currying*, named after the logician Haskell Curry, a pioneer of lambda calculus, the mathematical basis of functional programming languages.