

## Symbols and Lists

- Symbols
- Constructing Lists with Cons
- Constructing List Literals
- Creating Lists with `list`
- Nested Lists
- Taking lists apart
- Common list procedures
  - `length`
  - `reverse`
  - `append`
  - `list-ref`

## Symbols

While your initial exercises in Scheme have been numeric, Scheme is not limited to numerical computation, but can also operate on pure symbols.

Scheme's ancestor, Lisp, was originally developed to aid in experiments in artificial intelligence. At the time, a leading theory suggested that intelligence emphasizes symbolic manipulation. Hence, it is sensible that Lisp and Scheme include symbols as a basic type.

When we want to refer to something as a value involved in a computation, rather than as the name of some other value, we put an apostrophe (usually pronounced “quote”) in front of it. In effect, by quoting the symbol, we're telling Scheme to take it literally and without further interpretation or evaluation:

```
> 'sample
sample
```

Note that you can quote many different things. You can even quote Scheme expressions.

```
> '(+ 2 3)
(+ 2 3)
```

## Constructing Lists with Cons

In addition to “unstructured” data types such as symbols and numbers, Scheme supports *lists*, which are structures that contain other values as elements. There is one list, the *empty list*, that contains no elements at all. Any other list is constructed by attaching some value, called the *car* of the new list, to a previously constructed list, which is called the *cdr* of the new list.

Scheme's name for the empty list is a pair of parentheses with nothing between them: `()`. When we refer to the empty list in a Scheme program, we have to put an apostrophe before the left parenthesis, so that Scheme won't mistake the parentheses for a procedure call:

```
> '()  
( )
```

Since this conventional name for the empty list is not very readable, our implementation of Scheme also provides a built-in name, `null`, for the empty list. We follow this usage and recommend it.

```
> null  
( )
```

The “constructor” procedure for non-empty lists is called `cons`. It takes two arguments and returns a list that is just like the second argument, except that the first argument has been added at the beginning, as a new first element. By repeated applications of `cons`, we can build up a list of any size:

```
> (define singleton (cons 'sample null))  
> singleton  
(sample)  
> (define doubleton (cons 'another-element singleton))  
> doubleton  
(another-element sample)  
> (define tripleton (cons 'yet-another-element doubleton))  
> tripleton  
(yet-another-element another-element sample)  
> (cons 'senior (cons 'junior (cons 'sophomore (cons 'freshling null))))  
(senior junior sophomore freshling)
```

The `cons` procedure never returns an empty list, since it always adds an element at the beginning of another list.

## Constructing List Literals

As you may have noted from the discussion of atoms, there is another way to create lists. You can

- write out a literal constant -- a numeral or a symbol -- for each datum
- separating the elements with spaces
- enclose the whole thing in parentheses
- attach an apostrophe at the beginning.

For example, the value of the expression

```
'(38 72 apple -1/3 sample)
```

is a five-element list consisting of two numbers, a symbol, another number, and finally another symbol. Note that the apostrophe blocks the evaluation of the whole list, so that it is not necessary to quote separately the symbols that occur as elements of the list.

In a *list literal* like this one, the apostrophe must be present so that Scheme does not misinterpret the left parenthesis as the beginning of a procedure call. Sometimes that apostrophe is all that distinguishes two different, correctly formed expressions. For instance, `(+ 5 3)` is a procedure call that has the value 8, whereas `'(+ 5 3)` is a list literal denoting a list of three elements -- the symbol `+` and the numbers 5 and 3.

```
>
> (+ 5 3)
8
> '(+ 5 3)
(+ 5 3)
```

While list literals seem like a convenient way to create lists, experience shows that they can also lead to problems. We recommend that you generally avoid using list literals (and instead use the next strategy to create longer lists).

## Creating Lists with `list`

Yet another way to create a list is to invoke a procedure named `list`. This procedure takes all of its arguments, however many of them there may be, and packs them into a list. (Behind the scenes, `list` invokes `cons` once for each element of the completed list, to hook that element onto the previously created `cdr`.) Just as the addition procedure `+` sums its arguments and returns the result, so the `list` procedure collects its arguments and returns the resulting list:

```
> (list 38 72 'apple -1/3 'sample)
(38 72 apple -1/3 sample)
> (define a 2)
> (define b 3)
> (list a b)
(3 10)
```

## Nested Lists

It is possible, and indeed common, for a list to be an element of another list. For instance, the expression

```
(list 'alpha 'beta (list 'gamma-1 'gamma-2) 'delta)
```

creates a *four-element* list: Its first element is the symbol `alpha`, its second is the symbol `beta`, its third is a two-element list comprising the symbols `gamma-1` and `gamma-2`, and its fourth is the symbol `delta`.

It is possible for all of the elements of a list to be lists. It is possible for a list that is an element of another list to have lists as its elements, and so on -- lists can be embedded within lists to any desired level of nesting. This idea is subtler and more powerful than it may initially seem to be.

## Taking lists apart

To recover elements from a list, one commonly uses the built-in Scheme procedures `car`, which takes one argument (a non-empty list) and returns its first element, and `cdr`, which takes one argument (a non-empty list), and returns a list just like the one it was given, except that the first element has been removed. In a sense, `car` and `cdr` are the inverses of `cons`; if you think of a non-empty list as having been assembled by a call to the `cons` procedure, `car` gives you back the first argument to `cons` and `cdr` gives you back the second one.

```
> (car (cons 'apple (cons 'orange null)))
apple
> (cdr (cons 'apple (cons 'orange null)))
(orange)
```

If you want the second rather than the first element of a list, you can combine `car` and `cdr` to extract it:

```
> (define sample (cons 'apple (cons 'orange null)))
> (car (cdr sample))
orange
```

The idea is that the procedure call `(cdr sample)` computes a list just like `sample` except that the symbol `apple` is gone, and then `car` gives you the first element of that computed list. Similarly, `(car (cdr (cdr longer-list)))` is the third element of `longer-list`, and so on.

## Common list procedures

Just as Scheme provides many built-in procedures that perform simple operations on numbers, there are several built-in procedures that operate on lists. Here are four that are very frequently used:

### **length**

The `length` procedure takes one argument, which must be a list, and computes the number of elements in the list. (An element that happens to be itself a list nevertheless contributes 1 to the total that `length` computes, regardless of how many elements it happens to contain.)

### **reverse**

The `reverse` procedure takes a list and returns a new list containing the same elements, but in the opposite order.

```
> (reverse '(a b c))
(c b a)
```

## **append**

The `append` procedure takes any number of arguments, each of which is a list, and returns a new list formed by stringing together all of the elements of the argument lists, in order, to form one long list.

## **list-ref**

The `list-ref` procedure takes two arguments, the first of which is a list and the second a non-negative integer less than the length of the list. It recovers an element from the list by skipping over the number of initial elements specified by the second argument (applying `cdr` that many times) and extracting the next element (by invoking `car`). So `(list-ref sample 0)` is the same as `(car sample)`, `(list-ref sample 1)` is the same as `(car (cdr sample))`, and so on.