

Lists in Scheme

Summary: We consider lists, the most important compound data type in Lisp.

Contents:

- Introducing Lists
- Creating Lists with `list`
- Constructing Lists with `cons`
- Warning: Cons-ing to Non-Lists
- Constructing List Literals
- Nested Lists
- Taking Lists Apart
- Common List Procedures
 - `length`
 - `reverse`
 - `append`
 - `list-ref`

Introducing Lists

One of Scheme's great strengths is that it includes a powerful data structure, the list, as a built-in data type. In contrast to Scheme's "unstructured" data types, such as symbols and numbers, lists are "structures" that contain other values as elements. **A list is an ordered collection of values.** In Scheme, lists can be *heterogeneous*, in that they may contain different kinds of values.

As we will see throughout the semester, lists provide an elegant and easy-to-use mechanism for organizing collections of information.

In order to work with lists, you need to know how to create lists, update lists (or at least create new lists based on other lists), and extract values from lists.

Creating Lists with `list`

The easiest 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. 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)
(2 3)
> (list 'a a 'b b)
(a 2 b 3)

```

Constructing Lists with cons

Although `list` is a convenient way to build lists, it is actually built from other operations. Behind the scenes, `list` invokes `cons` once for each element of the completed list, to hook that element onto the previously created list.

We have to start somewhere. As you might guess, the simplest list is 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
()

```

If, for some reason, the implementation of Scheme that you're using does not include `null`, you can always define it yourself.

```

(define 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 'third-year (cons 'second-year (cons 'freshling null))))
(senior third-year second-year freshling)

```

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

Warning: Cons-ing to Non-Lists

Although we have defined `cons` in such a way that it expects a list as its second parameter, it is possible to use a value other than a list as the second parameter. When you do so, you build something that is a bit like a list, but also a bit different. Most Scheme environments indicate the difference by putting a period before the final value.

```
> (cons 'a (cons 'b null))
(a b)
> (cons 'a (cons 'b 'null))
(a b . null)
```

If you are trying to create a list and you see the period, you have probably done something wrong, so you'll need to think about your algorithm.

Constructing List Literals

Warning! This section discusses a technique that we do not recommend using. We include it because it reveals a lot about the workings of Scheme.

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,
- separate the elements with spaces,
- enclose the whole thing in parentheses, and
- place an apostrophe (a single quote) 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.

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.

Many implementations of Scheme provide these variants, using names like `cadr` (for the car of the cdr) and `cddr` for the (cdr of the cdr).

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.

Copyright © 2006 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.