

Association Lists

- Representing Databases
- `assoc`, Scheme's built-in lookup procedure
- Extracting Information
- Using More Complex Records
- Using Other Keys
- Related Procedures

Consider the organization of a simple telephone directory for on-campus telephones: a sequence of entries, each consisting of a name and a four-digit telephone number. In Scheme, it's natural to use strings for names; it turns out that telephone numbers should also be represented as strings, since string operations make a useful kind of sense when applied to telephone numbers and integer operations do not. (For instance, `(string-append "269-" extension)` does something useful if the value of `extension` is a string, but not if it is an integer.)

Representing Databases

To represent each individual entry in a telephone directory, we can use a list, such as `("Henry Walker" "4208")`, `("John Stone" "3181")`, or `("Sam Rebelsky" "4410")`, with the name as the `car` of the entry and a list of the telephone number as the `cdr`. An entire directory, then, would be a list of such entries:

```
;;; Value:
;;; science-chairs-directory
;;; Type:
;;; List of lists.
;;; Each sublist is of length two and contains a name and a phone number.
;;; Both of those values are strings.
;;; Contents:
;;; A list of the department and division chairs in the Science division
;;; in academic year 2000-2001.
(define science-chairs-directory
  (list (list "Bruce Voyles" "3038")
        (list "Diane Robertson" "3039")
        (list "Martin Minelli" "3007")
        (list "Emily Moore" "4201")
        (list "Paul Tjossem" "1234")
        (list "David Lopatto" "3168")))
```

In Scheme, a list of pairs or lists is called an *association list* or *alist*.

As the telephone-directory example illustrates, a particularly common application of association lists involves looking for a desired name or first component of a pair and retrieving the second component of a pair. Thus, the first component of each pair (the `car` of a pair) often is called a *key*, and the `cdr` of the pair is its *associated data* or *value*. For example, in the above illustration, "Martin Minelli", "Emily Moore", and "David Lopatto" are some of the keys, and the lists that contain only

telephone numbers are the associated data. Thus an association list is a simple way to implement a small database.

assoc, Scheme's built-in lookup procedure

Since such applications are very common, Scheme provides procedures to retrieve from an association list the pair containing a specified key. The most frequently used procedure of this kind is `assoc`. Given a key and association list, `assoc` returns the first pair with the given key. If the key does not occur in the association list, then `assoc` returns `#f`. For example, the value of `(assoc "Emily Moore" science-chairs-directory)` is `("Emily Moore" "4201")`, while the value of `(assoc "Sam Rebelsky" science-chairs-directory)` is `#f`.

Extracting Information

To find the telephone number corresponding to a given name, we could apply the `cadr` procedure to the result of `assoc`:

```
;;; Procedure:
;;; look-up-telephone-number
;;; Parameters:
;;; name, a string
;;; directory, a list of telephone book entries
;;; Purpose:
;;; Looks up someone's telephone number in the directory.
;;; Produces:
;;; A phone number, if that person is in the directory.
;;; The symbol unlisted if the person is not in the directory.
;;; Preconditions:
;;; The name must be a string. [Unverified]
;;; Each telephone book entry must be a list. [Unverified]
;;; Element 0 of each telephone book entry must be a string which
;;; represents a name. [Unverified]
;;; Element 1 of each telephone book entry must be a string which
;;; represents that person's phone number. [Unverified]
;;; Postconditions:
;;; If an entry for name appears somewhere in the directory, returns
;;; the corresponding phone number.
;;; If multiple entries with the same name appear, returns one of them.
;;; If no entries appear, returns false (#f).
;;; Does not affect the directory.
(define look-up-telephone-number
  (lambda (name directory)
    (if (assoc name directory)
        (cadr (assoc name directory))
        'unlisted)))
```

The value of the call

```
(look-up-telephone-number "Emily Moore" science-chairs-directory)
```

is "4201" and the value of

```
(look-up-telephone-number "Sam Smith" science-chairs-directory)
```

is the symbol `unlisted`.

Note that the result depends on the directory. For example,

```
(look-up-telephone-number "Emily Moore" null)
```

is the symbol `unlisted`.

Some of you may recall asking why `if` might take a value other than `#t` or `#f` as a parameter. This procedure is one example of why.

Using More Complex Records

In the previous example, we have only one value (the phone number) associated with a key. However, in practice, we often want to associate many values with the same key. For example, we might want to note which department each chair is associated with. Here's a new version of our previous list that also includes that information.

```
;;; Value:
;;; science-chairs-directory
;;; Type:
;;; List of lists.
;;; Each sublist is of length two and contains a name and a phone number.
;;; Both of those values are strings.
;;; Contents:
;;; A list of department and other chairs in the Science division in
;;; academic year 2000-2001.
(define science-chairs-directory
  (list (list "Bruce Voyles" "3038" "Science")
        (list "Diane Robertson" "3039" "Biology")
        (list "Martin Minelli" "3007" "Chemistry")
        (list "Emily Moore" "4201" "Math/CS")
        (list "Paul Tjossem" "1234" "Physics")
        (list "David Lopatto" "3168" "Psychology")))
```

You should note a few things about this list. First, we've left the phone number as element 1 so that `look-up-telephone-number` still works. Second, we've taken advantage of Scheme's decision to ignore spaces between values by using spaces to put stuff in more tabular form.

Using Other Keys

The `assoc` procedure works fine if the key is the first element of a data item. But what if it's the second (or third, or fourth, or ...). For example, what if we know someone's phone number and want to find his or her name? Then we can't rely on `assoc`, because it only looks at the first element of each list. Instead, we need to write our own procedure. For example, to find someone with a particular phone number, we might write:

```

;;; Procedure:
;;;   look-up-by-number
;;; Parameters:
;;;   number, a string
;;;   directory, a list of telephone book entries
;;; Purpose:
;;;   Looks up the entry for a particular phone number.
;;; Produces:
;;;   A telephone book entry, if the number is listed.
;;;   False (#f) if the number is not listed.
;;; Preconditions:
;;;   The number must be a string. [Unverified]
;;;   Each telephone book entry must be a list. [Unverified]
;;;   Element 1 of each telephone book entry must be a string which
;;;     represents that person's phone number. [Unverified]
;;; Postconditions:
;;;   If a phone number appears in the directory, returns the
;;;     corresponding entry.
;;;   If the phone number appears in multiple entries, returns
;;;     one of those entries.
;;;   If no entries with that phone number appear, returns false (#f).
;;;   Does not affect the directory.
(define look-up-by-number
  (lambda (number directory)
    (cond
      ; If there are no entries in the directory, our desired
      ; entry is not there.
      ((null? directory) #f)
      ; If the number we're looking for is in the initial entry,
      ; use that entry
      ((equal? number (list-ref (car directory) 1))
       (car directory))
      ; Otherwise, look in the rest of the directory.
      (else (look-up-by-number number (cdr directory))))))

```

Related Procedures

The `assoc` procedure is actually one of three related built-in procedures in Scheme; the other two are `assq` and `assv`. Each of these procedures scan association lists for keys. They differ only in the test used for determining when a key is found:

- `assoc` uses the predicate `equal?` to compare the key sought with the key components of the entries in the association list.
- `assq` uses the predicate `eq?` for those comparisons.
- `assv` uses the predicate `eqv?` for those comparisons.

You may wish to refresh your memory on the purpose of these predicates (hint hint).