

Association Lists

Summary: We consider a number of Scheme procedures that can be used to look up information.

Contents:

- Introduction
- Representing Database Tables
- `assoc`, Scheme's Built-in Lookup Procedure
- Extracting Information
- Looking Up a Telephone Number, Revised
- Using More Complex Records
- Implementing `assoc`
- Using Other Keys
- Related Procedures

Procedures covered in this reading:

- `(assoc key alist)`
- `(assq key alist)`
- `(assv key alist)`

Introduction

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 is 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 Database Tables

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 cadr. An entire directory, then, would be a list of such entries:

```
;;; Value:
;;;  science-chairs-directory
;;; Type:
;;;  List of lists.
;;;  Each sublist is of length three and contains a name and a phone number.
;;;  Each of those values are strings.
;;; Contents:
```

```

;;; A list of the department and division chairs in the Science division
;;; in academic year 2006-2007.
(define science-chairs-directory
  (list (list "Samuel A. Rebelsky" "4410")
        (list "Vince Eckhart" "4354")
        (list "Lee Sharpe" "3008")
        (list "Henry Walker" "4208")
        (list "Mark Chamberland" "4207")
        (list "Bob Cadmus" "3016")
        (list "Janet Gibson" "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 list and retrieving another 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, "Samuel A. Rebelsky", "Mark Chamberland", and "Janet Gibson" are some of the keys, and the lists that contain department and telephone numbers are the associated data. Thus, an association list is a simple way to implement a small table in a 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,

```

> (assoc "Samuel A. Rebelsky" science-chairs-directory)
("Samuel A. Rebelsky" "Science" "4410")
> (assoc "Russell King Osgood" science-chairs-directory)
#f

```

Extracting Information

To find the telephone number corresponding to a given name, we could apply the `cadr` procedure (or `list-ref` with a position of 1). Here's an example.

```

;;; Procedure:
;;; lookup-telephone-number
;;; Parameters:
;;; name, a string
;;; directory, a list of telephone book entries
;;; Purpose:
;;; Looks up someone's telephone number in the directory.
;;; Produces:
;;; number, a string
;;; Preconditions:
;;; 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 the string "unlisted"
;;; Does not affect the directory.
(define lookup-telephone-number
  (lambda (name directory)
    (if (assoc name directory)
        (cadr (assoc name directory))
        "unlisted")))

```

For example,

```

> (lookup-telephone-number "Samuel A. Rebelsky" science-chairs-directory)
"4410"
> (lookup-telephone-number "Russell King Osgood" science-chairs-directory)
"unlisted"

```

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

```

> (lookup-telephone-number "Samuel A. Rebelsky" null)
"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.

Looking Up a Telephone Number, Revised

One problem with the `lookup-telephone-number` procedure given above is that it calls `assoc` twice, once to determine whether the phone number is in the list and once to determine the phone number. Rather than calling `assoc` twice, we might call it once and send the result to a helper procedure.

```

(define lookup-telephone-number
  (lambda (name directory)
    (let ((assoc-result (assoc name directory)))
      (if assoc-result
          (cadr assoc-result)
          "unlisted"))))

```

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 three and contains a name, a phone number,
;;;   and a division.
;;;   All three of those values are strings.
;;; Contents:
;;;   A list of department and other chairs in the Science division in
;;;   academic year 2006-2007.
(define science-chairs-directory
  (list (list "Samuel A. Rebelsky" "4410" "Science")
        (list "Vince Eckhart" "4354" "Biology")
        (list "Lee Sharpe" "3008" "Chemistry")
        (list "Henry Walker" "4208" "Computer Science")
        (list "Mark Chamberland" "4207" "Mathematics and Statistics")
        (list "Bob Cadmus" "3016" "Physics")
        (list "Janet Gibson" "3168" "Psychology")))

```

You should note a few things about this list. First, we've left the phone number as element 1 so that `lookup-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.

Implementing `assoc`

As is the case with many of the built-in Scheme procedures, `assoc` is relatively easy to implement ourselves. In essence, `assoc` recursively steps through the list until it finds a match or runs out of elements.

```

(define assoc
  (lambda (key alist)
    (cond
      ; If there are no entries left in the association list,
      ; there are no entries with the given key.
      ((null? alist) #f)
      ; If the key we're looking for is the key of the first
      ; entry, then use that entry.
      ((equal? key (car (car alist))) (car alist))
      ; Otherwise, look in the rest of the association list.
      (else (assoc key (cdr alist))))))

```

The DrScheme implementation of `assoc` also includes some error checking that you'll learn how to do later this semester.

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:
;;; lookup-by-number
;;; Parameters:
;;; number, a string
;;; directory, a list of telephone book entries
;;; Purpose:
;;; Looks up the entry for a particular phone number.
;;; Produces:
;;; entry, a telephone book entry.
;;; Preconditions:
;;; 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 an entry with phone number number appears in the directory,
;;; (1) number equals (list-ref entry 1);
;;; (2) entry is an element of directory
;;; If no entries with that phone number appear, entry is false (#f).
;;; Does not affect the directory.
(define lookup-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 (lookup-by-number number (cdr directory))))))

```

It might even make sense to generalize this procedure, so that we can look up by name, by telephone number, or by department.

```

;;; Procedure:
;;; lookup
;;; Parameters:
;;; pos, a nonnegative integer [unverified]
;;; key, a string [unverified]
;;; directory, a list of lists [unverified]
;;; Purpose:
;;; Looks up the entry that corresponds to a particular key.
;;; Produces:
;;; entry, one of the member lists.
;;; Preconditions:
;;; Each element of directory must be a list. [Unverified]
;;; Each element of directory must have at least (pos+1) values. [Unverified]
;;; Postconditions:
;;; If there is a list, lst, in directory such that key equals
;;; (list-ref lst pos), then entry is some such list.
;;; Otherwise, entry is #f.
;;; Does not affect the directory.
(define lookup
  (lambda (pos key directory)
    (cond

```

```

((null? directory) #f)
((equal? key (list-ref (car directory) pos))
 (car directory))
(else (lookup pos key (cdr directory))))))

```

We can now define `lookup-by-number` (and even `lookup-by-department` and `lookup-by-name`) in terms of `lookup`.

```

(define lookup-by-number
  (lambda (number directory)
    (lookup 1 number directory)))
(define lookup-by-department
  (lambda (department directory)
    (lookup 2 department directory)))
(define lookup-by-name
  (lambda (name department)
    (lookup 0 name directory)))

```

And we can see that these work fine.

```

> (lookup-by-department "Biology" science-chairs-directory)
("Vince Eckhart" "4354" "Biology")
> (lookup-by-department "Chemistry" science-chairs-directory)
("Lee Sharpe" "3008" "Chemistry")
> (lookup-by-department "Economics" science-chairs-directory)
#f
> (lookup-by-number "4410" science-chairs-directory)
("Samuel A. Rebelsky" "4410" "Science")
> (lookup-by-number "3000" science-chairs-directory)
#f

```

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).

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.