**13. Association Lists and Hash Tables**

**1. Introduction**

In this chapter, I will explain association lists and hash tables which are data types to represent associations. Associated data are pairs of keys and values and the value is defined by the key uniquely. Table 1 shows pairs of books and authors. Authors can be defined by books, while books cannot be defined by authors because a same author writes several books. In the Table 1, as P. Graham and L. Carroll wrote two books, their book cannot be defined by their name.

Table 1: Authors and books.

|  |  |
| --- | --- |
| **Author** | **Book** |
| P. Graham | On Lisp |
| P. Graham | ANSI Common Lisp |
| E. S. Raymond | The Cathedral and the Bazaar |
| K. Dybvig | The Scheme Programming Language |
| F. P. Brooks, Jr. | The Mythical Man-Month |
| L. Carroll | Alice's Adventures in Wonderland |
| L. Carroll | Through the Looking-Glass, and What Alice Found There |

The association lists are available in all Scheme implementations as it is defined in the R5RS. But the search using the association lists is slow (order of*O(n)*). Using hash tables is better in speed (order of *O(1)*), but it is not defined in the R5RS and it is implementation dependent. The MIT-Scheme has one. If your favorite implementation doesn't, you can define one by yourself (see <http://www.math.grin.edu/~stone/events/scheme-workshop/hash-tables.html>).

**2. Association Lists**

The association list is a list consisting of pairs and it is a basic data type to represent association. Symbols, characters, and numbers are often used as keys because they can be compared using fast comparison functions such as eq? or eqv?. Strings should be converted to symbols before used as keys for better performance.

Followings are examples of association lists. Association lists should consist of either dot-pairs or ordinal lists.

'((hi . 3) (everybody . 5) (nice . 3) (to . 10) (meet . 4) (you . 8))

'((1 2 3) (4 5 6) (7 8 9))

Functions **assq**, **assv**, and **assoc** are to search an item from an association list. These functions search the association list from the beginning step by step. They return the pair whose car is equal to the given key if they find. If not the functions return #f. The functions use eq?, eqv?, and equal? to compare keys, respectively, which means that function assq is the fastest and assoc the slowest. This shows that strings, vectors, and lists should be converted to symbols or numbers (if possible) to improve the performance.

In general [hash tables](http://www.shido.info/lisp/scheme_ah_e.html#hash) are better to search from mass of data.

Followings show examples of search from association lists.

(define wc '((hi . 3) (everybody . 5) (nice . 3) (to . 10) (meet . 4) (you . 8)))

⇒ wc

(assq 'hi wc)

⇒ (hi . 3)

(assq 'you wc)

⇒ (you . 8)

(assq 'i wc)

⇒ ()

(define n '((1 2 3) (4 5 6) (7 8 9)))

⇒ n

(assv 1 n)

⇒ (1 2 3)

(assv 8 n)

⇒ ()

**3. Hash Tables**

[Hash tables](http://en.wikipedia.org/wiki/Hash-table) are data type that convert keys to integers by a hash function and that store values at the addresses indicated by the integers. When the table is uncrowded enough, searching, inserting, and updating can be done in the order of *O(1)*. Following shows some basic functions about hash tables defined in the MIT-Scheme. See [MIT-Scheme Manual](http://www.swiss.ai.mit.edu/projects/scheme/documentation/scheme_12.html#SEC119) for detailed information.

**(make-eq-hash-table size)**,  
**(make-eqv-hash-table size)**,  
**(make-equal-hash-table size)**,  
**(make-string-hash-table size)**

These functions create a hash table. They use eq?, eqv?, equal?, and string=? to compare keys, respectively. The initial size of the hash table (**size**) can be specified (optional). The eq-hash-table is the fastest because it just compares addresses of keys. The equal-hash-table and string-hash-table are slower because their keys are sequence.

**(hash-table/put! hash-table key datum)**

It sets the value associated with the **key** in the **hash-table** to the **datum**.

**(hash-table/get hash-table key default)**

It returns the associated value with the **key** in the **hash-table**. If the **key** is not in the **hash-table**, it returns the **default**.

**(hash-table->alist hash-table)**

It converts the **hash-table** to an association list.

**4. Making Passwords**

Let's write a password creating program as an example of association lists and hash tables.

Passwords taken from a dictionary can be broken easily and on the other hand, totally random passwords are extremely difficult to remember and input. The program creates ten passwords with partially possible spelling. Password should be changed as frequent as possible, but I am lazy to make passwords by myself. Using this program, I can change the password easily.

The program consists of two part. One is to create the data of frequency of successive characters (stat-spell.scm) and the other to make passwords based on the data (make-pw.scm).

**4.1. stat-spell.scm**

This program reads English sentences and associates characters with the frequency of the following characters. The data is stored as a hash table and converted to an associated list to be output to a file (stat-spell.dat) at the end. [code 1] shows the source code.

[code 1]

01: ;;; make an alist of probable spelling from a given English text

02:

03: (define (skip-char? c)

04: (or (not (char-graphic? c)) (memv c '(#\: #\; #\' #\" #\`))))

05:

06: (define (ss-make-alist c alist)

07: (let ((p (assv c alist)))

08: (if p

09: (begin

10: (set-cdr! p (1+ (cdr p)))

11: alist)

12: (cons (cons c 1) alist))))

13:

14: (define (ss-make-dat filename)

15: (let ((char-hash (make-eqv-hash-table)))

16: (with-input-from-file filename

17: (lambda ()

18: (let loop ((c #\Space))

19: (let ((c1 (read-char)))

20: (if (not (eof-object? c1))

21: (if (skip-char? c1)

22: (loop c)

23: (let ((c1 (char-downcase c1)))

24: (hash-table/put! char-hash c

25: (ss-make-alist c1 (hash-table/get char-hash c '())))

26: (loop c1))))))))

27: (with-output-to-file "stat-spell.dat"

28: (lambda ()

29: (display "(define \*stat-spell\* \'(")

30: (newline)

31: (let loop ((alst (sort (hash-table->alist char-hash)

32: (lambda (x y) (char33: (if (pair? alst)

34: (begin

35: (write (car alst))

36: (newline)

37: (loop (cdr alst)))))

38: (display "))")

39: (newline)))))

**(skip-char? c)**

It returns #t if **c** is not a graphic character or **c** is #\:, #\;, #\', or #\". These characters are skipped when the text is read.

**(ss-make-alist c alist)**

It takes two arguments; the association list of frequency of character (**alist**) and a character (**c**). If the **c** is in the **alist**, it increments the cdr part of the pair. If not, it returns   
(cons (cons **c** 1) **alist**). This function uses set-cdr!.

**(ss-make-dat filename)**

It reads characters from the file named **filename** and associates the characters with the association list of frequency of the following characters. It stores the result to a file [stat-spell.dat](http://www.shido.info/lisp/stat-spell.dat) as an association list. On lines 34 and 35, it updates the association list of frequency stored in the hash table. The final data stored in the stat-spell.dat is an association list of association lists. For instance,

(#\v (#\y . 1) (#\a . 3) (#\o . 7) (#\e . 51) (#\i . 15))

indicates that #\y, #\a, #\o, #\e, and #\i follow #\v for 1, 3, 7, 51, and 15 times, respectively.

**4.2. make-pw.scm**

It creates ten passwords based on the stat-spell.dat. The procedure is as follows:

1. Creating lists of strings consisting of 9 – 13 characters randomly selected based on the frequency data. The character #\Space is added at the end of the list.
2. Adding a random number in a range of 00 – 99 at the end of the list of randomly selected characters.
3. Converting #\Space to #\-, #\\_, #\/, #\Space, #\., or #\, randomly.
4. Capitalizing 30% of alphabetic characters randomly.

[code 2] shows the source. The function random is not defined in the R5RS. (So it is implementation depending.)

[code 2]

01: ;;; make password from the alist of probable spelling

02:

03: (load "stat-spell.dat") ; \*stat-spell\* (alist for following characters) is in.

04:

05: (define (alist->hash al mode)

06: (let ((h (case mode

07: ((eq) (make-eq-hash-table))

08: ((eqv) (make-eqv-hash-table))

09: ((equal) (make-equal-hash-table))

10: ((string) (make-string-hash-table)))))

11: (for-each (lambda (p)

12: (hash-table/put! h (car p) (cdr p)))

13: al)

14: h))

15:

16: (define \*stat-spell-hash\* (alist->hash \*stat-spell\* 'eqv))

17:

18: (define (pw-random-select vec)

19: (vector-ref vec (random (vector-length vec))))

20:

21: (define (random00)

22: (let loop ((i 0) (acc '()))

23: (if (= i 2)

24: (list->string acc)

25: (loop (1+ i) (cons (pw-random-select '#(#\0 #\1 #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9)) acc)))))

26:

27: (define (occasional-upcase c)

28: (if (< (random 10) 3)

29: (char-upcase c)

30: c))

31:

32: (define (pw-enhance ls)

33: (list->string

34: (map (lambda (c)

35: (cond

36: ((char=? c #\Space)

37: (pw-random-select '#(#\- #\\_ #\/ #\Space #\. #\, #\@ #\? #\( #\))))

38: ((char-alphabetic? c)

39: (occasional-upcase c))

40: (else c)))

41: (cdr (reverse! ls)))))

42:

43:

44: (define (random-following alist)

45: (let ((n (random (apply + (map cdr alist)))))

46: (let loop ((j 0) (alist alist))

47: (if (pair? alist)

48: (let\* ((pair (car alist))

49: (k (+ j (cdr pair))))

50: (if (> k n)

51: (car pair)

52: (loop k (cdr alist))))))))

53:

54: (define (make-pw h n)

55: (let loop ((i 0) (c #\Space) (acc '()))

56: (if (= i n)

57: (string-append

58: (pw-enhance (cons #\Space (cons c acc)))

59: (random00))

60: (loop (1+ i)

61: (random-following (hash-table/get h c '((#\Space . 1))))

62: (cons c acc)))))

63:

64: (define (pw-candidates)

65: (let loop ((i 0))

66: (if (< i 10)

67: (begin

68: (display i)

69: (display ": ")

70: (write (make-pw \*stat-spell-hash\* (+ 9 (random 4))))

71: (newline)

72: (loop (1+ i)))

73: 'done)))

**(alist->hash al mode)**

Converting an association list (**al**) to a hash table with **mode**.

**(pw-random-select vec)**

Selecting the item randomly from a vector (**vec**). Vectors will be explained in the following chapter. Vectors suit for random access rather than lists.

**(random00)**

Creating a string "00" — "99" randomly.

**(occasional-upcase c)**

Capitalizing 30% of **c** randomly.

**(pw-enhance ls)**

Making a password stronger by processing a list of characters (**ls**).

**(random-following alist)**

Selecting a character randomly from a association list (**alist**) based on the frequency.

**(make-pw h n)**

Creating a password consisting of (**n+3**) characters using the hash table (**h**). The password ends with random two digits.

**(pw-candidates)**

Creating ten passwords consisting of 12 – 15 characters.

Following shows how to use the program. Ten passwords are created. The stat-spell is required only once. Load make-pw and execute (pw-candidate) to create passwords.

(compile-file "stat-spell.scm")

(compile-file "make-pw.scm")

(load "stat-spell")

(load "make-pw")

;;; creating spelling data according to sicp\_foreword.txt

(ss-make-dat "sicp\_foreword.txt")

;;; making ten passwords using the spelling data.

(pw-candidates)

0: "TH-haVEMis/t/19"

1: "MoghaTrad\_-Z.82"

2: "seatSk-nev 64"

3: "wIE t incar,92"

4: "<bIcoMpure,e/48"

5: "d\_t,ustampRE.48"

6: "wiN,O/LEx.90"

7: "OfU-D wogr/29"

8: "d the.ole\_95"

9: "tippRiooU,77"

;Value done

**5. Summary**

You can download the password creating program from [here](http://www.shido.info/lisp/scheme_ah.zip).

I will explain about vectors in the next chapter.