**14. Vectors and Structures**

**1. Introduction**

In this chapter, I am going to explain about vectors and structures.

Vectors are set of data indexed by integer. Different types of data can be stored in a same vector, which is different from arrays of the C language. Vectors are more compact than lists and the access time is shorter. On the other hand, as vectors are manipulated using side effects, it may cause bags.

Structures in the Scheme is similar to those in the C language. However, the structures in the Scheme is more convenient than those of the C language, because the Scheme writes accessors and setters for the structures automatically, which is one of the benefits of macro of the Lisp/Scheme programming languages.

**2. Vectors**

**2.1. Literals**

Vectors are represented by '**#(**' and '**)**' like #(1 2 3). They should be quoted when used as literals

Examples:

'#(1 2 3) ; a vector of integers

'#(a 0 #\a) ; a vector consisting of a symbol, an integer, and a character

**2.2. Functions for vectors**

Following functions are defined in the R5RS.

**(vector? obj)**

It returns #t if **obj** is a vector.

**(make-vector k)  
(make-vector k fill)**

It returns a vector with **k** elements. If the second argument (**fill**) is specified, each elements are initialized with **fill**.

**(vector obj ...)**

It returns a vector consisting of the arguments.

**(vector-length vector)**

It returns the length of the **vector**.

**(vector-ref vector k)**

It returns the **k**-th element of the **vector**.

**(vector-set! vector k obj)**

It sets the **k**-th element of **vector** to **obj**.

**(vector->list vector)**

It converts **vector** to a list.

**(list->vector list)**

It converts **list** to a vector.

**(vector-fill! vector fill)**

It sets all the items of **vector** to **fill**.

Example: A function that calculate the sum of vectors.

01: (define (vector-add v1 v2)

02: (let ((lenv1 (vector-length v1))

03: (lenv2 (vector-length v2)))

04: (if (= lenv1 lenv2)

05: (let ((v (make-vector lenv1)))

06: (let loop ((i 0))

07: (if (= i lenv1)

08: v

09: (begin

10: (vector-set! v i (+ (vector-ref v1 i) (vector-ref v2 i)))

11: (loop (1+ i))))))

12: (error "different dimensions."))))

**Exercise 1**

Write a function that calculates the inner product of two vectors.

**3. Structures**

**3.1. General Feature**

While structures are not defined in the R5RS, [Structures similar to those of the Common Lisp](http://www.h7.dion.ne.jp/~matsu/feature/common-lisp/data-structure/struct.html) is implemented in many Scheme implementations.

The reality of the structures are vectors. Each slot is named by using a [macro](http://www.shido.info/lisp/scheme_syntax_e.html), which I will explain in the next chapter (chapter 15). Structures express data with different kinds of attributes clearly. The macro that define the structure makes accessors and setters functions automatically. It is one of the greatest benefits of the Lisp/Scheme that you can write programs that write programs. By using this feature, you can write beautiful programs quickly.

**3.2. Structures in the MIT-Scheme**

In the MIT-Scheme, structures are defined by **define-structure**. I will explain using an example, because it is easier to understand.  
Let's think about books. Books have following attributes:

* title
* author(s)
* publisher
* published year
* ISBN

The structure book can be defined like as follows:

(define-structure book title authors publisher year isbn)

Following shows how to register ["The Cathedral and Bazaar"](http://www.oreilly.com/catalog/cathbazpaper/).

(define bazaar

(make-book

"The Cathedral and the Bazaar"

"Eric S. Raymond"

"O'Reilly"

1999

0596001088))

However, this way is inconvenient somehow because the association of attributes with values is not clear. The **keyword-constructor** option is available to solve this problem. Following code is the revised version using this option, in which the relation between attributes and values is clear. In addition, the order of attributes does not matter in this option. The **copier** option is available, which creates a copier function for the structure.

(define-structure (book keyword-constructor copier)

title authors publisher year isbn)

(define bazaar

(make-book

'title "The Cathedral and the Bazaar"

'authors "Eric S. Raymond"

'publisher "O'Reilly"

'year 1999

'isbn 0596001088))

* A function named **"[the name of structure]?"** checks if an object is an instance of the structure. For instance, check if the bazaar is an instance of thebook can be checked using book?:
* (book? bazaar)
* ;Value: #t
* The function named **"copy-[structure name]"** is to copy a structure. For instance, following is a code to copy bazaar to cathedral.
* (define cathedral (copy-book bazaar))
* The function named **"[structure name]-[attribute name]"** is to access to the value of the attribute. For instance, following shows how to access to thetitle of the bazaar.
* (book-title bazaar)
* ;Value 18: "The Cathedral and the Bazaar"
* The function named **"set-[structure name]-[attribute name]!"** is to set a value to the attribute. Updating the year of the bazaar to 2001 is done like as follows ( "The Cathedral and the Bazaar" has been revised in 2001).
* (set-book-year! bazaar 2001)
* ;Unspecified return value
* (book-year bazaar)
* ;Value: 2001

See [MIT/GNU Scheme Reference: 2.10 Structure Definitions](http://www.gnu.org/software/mit-scheme/documentation/scheme_3.html#SEC41) for further information on the structure.

**4. The Mastermind — A Simple Code Breaking Game**

I show a simple program of code breaking game as an example of vector. This is a game to guess the four-digit number of the opponent. The four-digit numbers consist of four **different** numerics in 0 – 9. The opponent should inform the guesser about how good the guess is by using numbers of 'bulls' and 'cows'.

* 1. The number of bull (Nbull) is a number of numerics whose value and position are correct.
  2. The number of cow (Ncow) is a number of numerics whose value is correct but wrong position.

For example, if the code is 5601 and the guess is 1685, the numbers of bulls and cows are 1 and 2.

The computer and the user guess the code of the opponent each other. Player who breaks the code with fewer guesses is the winner. The game is draw if both the user and computer break code at the same time.

**4.1. Expressing the four digit numbers**

The four digit numbers are expressed by vectors to calculate numbers of bulls and cows efficiently. This way of expression owes the property of the code that all digits should be different numerics.

Making a vector whose length is 10 and the value of each index (**k**) is set to the digit at that **k** appears. The four digits are counted as 1, 2, 3, and 4 from lower position. If the number does not appear, the value of the index is 0. For instance, 5601 and 1685 are represented like as follows:

5601 → #(2 1 0 0 0 4 3 0 0 0)

1685 → #(0 4 0 0 0 1 3 0 2 0)

In the case of 5601, as numeric 0, 1, 5, and 6 appear 2nd, 1st, 4th, and 3rd digit, respectively, the values of indexes 0, 1, 5, and 6 are 2, 1, 4, and 3 and the values of other indexes are 0 in its vector expression.

This expression allows fast comparison of two numbers. If the index of the two vectors are both positive and if the value is same, it is a bull, otherwise a cow. In the case of 5601 and 1685, as the values of index 6 are both 3 and indexes 1 and 5 are both positive, the values of bulls and cows are 1 and 2.

**4.2.The Design of the Program**

The design of the program is as follows:

* 1. The program makes a list consisting of the vector expressions of all four digit numbers with different numerics.
  2. The program selects one number from the list randomly.
  3. It shuffles the list of step (1).
  4. The program guesses the code of the user first and the user gives the number of the bulls and cows. Then the user guess the code of the program and the program shows the Nbull and Ncow.
  5. Step (3) is repeated until the number of bulls of the user or the computer becomes 4. If both numbers become 4 at the same time, the game is draw.

**4.3. The Source Code**

[code 1] shows the source code. It is long but not complicated very much. The game proceeds by a recursive function mastermind-rec.

[code 1]

01: ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

02: ;;;

03: ;;; mastermind.scm

04: ;;; by T.Shido

05: ;;;

06: ;;; User and computer try to locate the four-digit integer set by the opponents each other.

07: ;;; One who locates the integer with fewer question is the winner.

08: ;;; The four-digit integer contains four of numerals 0--9, like 0123, 3749 etc.

09: ;;; The opponents should tell the guesser

10: ;;; (1) number of numerals that are shared by the guessed and set numbers

11: ;;; at wrong position (cows)

12: ;;; and (2) number of numerals at collect position (bulls).

13: ;;;

14: ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

15: ;;;

16: ;;; The four-digit integers are represented by 10-cell vectors in the program

17: ;;; The value of n-th cell is the number of column that n appears in the integer.

18: ;;; in n is not appears the value is 0.

19: ;;; for example, 1234 is represented as #(0 4 3 2 1 0 0 0 0 0) and

20: ;;; 3916 as #(0 2 0 4 0 0 1 0 0 3).

21: ;;; With this inner representation, the score of the guess can be calculated faster.

22: ;;;

23: ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

24:

25:

26: ;;;

27: (define (1- x) (- x 1))

28:

29: ;;;

30: (define (char2int c)

31: (- (char->integer c) (char->integer #\0)))

32:

33: ;;; converting a list of 4 numbers to the vector notation

34: (define (ls2nvec ls)

35: (let ((vec (make-vector 10 0)))

36: (let loop ((i (length ls)) (ls ls))

37: (if (> i 0)

38: (begin

39: (vector-set! vec (car ls) i)

40: (loop (1- i) (cdr ls)))

41: vec))))

42:

43: ;;; converting the vector notation to string

44: (define (nvec2int vec)

45: (let loop ((i 0) (n 0))

46: (if (= i 10)

47: n

48: (let ((j (vector-ref vec i)))

49: (loop (1+ i) (+ n (if (> j 0)

50: (\* i (expt 10 (1- j)))

51: 0)))))))

52:

53: ;;;

54: (define (int2str i)

55: (string-append

56: (if (< i 1000) "0" "")

57: (number->string i)))

58:

59: ;;; reading integer from stdin

60: (define (read-integer str)

61: (string->number (read-from-stdin str)))

62:

63: ;;;

64: (define (read-from-stdin str)

65: (display str)

66: (newline)

67: (read-line))

68:

69: ;;;

70: (define (write-to-stdout . ls)

71: (for-each (lambda (obj) (display obj)) ls)

72: (newline))

73:

74: ;;; convert numeral string to the vector representation.

75: (define (str2nvec str)

76: (let ((vec (make-vector 10 0)))

77: (let loop ((i (string-length str)) (ls (string->list str)))

78: (if (pair? ls)

79: (begin

80: (vector-set! vec (char2int (car ls)) i)

81: (loop (1- i) (cdr ls)))

82: vec))))

83:

84: ;;; calculating the score of guess

85: (define (scoring vec0 vec1)

86: (let ((n (vector-length vec0)))

87: (let loop ((i 0) (score 0))

88: (if (< i n)

89: (let ((d0 (vector-ref vec0 i))

90: (d1 (vector-ref vec1 i)))

91: (loop (1+ i)

92: (+ score (if (and (< 0 d0) (< 0 d1))

93: (if (= d0 d1) 5 1)

94: 0))))

95: score))))

96:

97: ;;; show bulls and cows calculated from the score of user's guess

98: (define (show-user-score score)

99: (write-to-stdout "Number of bulls and cows in your guess:" )

100: (write-to-stdout "bulls: " (quotient score 5))

101: (write-to-stdout "cows: " (modulo score 5))

102: (newline))

103:

104: ;;; calculating the score of computer's guess from bulls and cows

105: (define (read-my-score gu0)

106: (write-to-stdout "My guess is: " (int2str (nvec2int gu0)))

107: (write-to-stdout "Give number of bulls and cows in my guess." )

108: (let ((na5 (\* 5 (read-integer "bulls: "))))

109: (+ na5 (read-integer "cows: ")))) ; the score is calculated by (5 \* bull + cow)

110:

111: ;;; reading the user guess

112: (define (read-user-guess)

113: (newline)

114: (str2nvec (read-from-stdin "Give your guess.")))

115:

116: ;;; shuffling the list of four-digit numbers

117: (define (shuffle-numbers ls0)

118: (let ((vec (list->vector ls0)))

119: (let loop ((n (vector-length vec)) (ls1 '()))

120: (if (= n 0)

121: ls1

122: (let\* ((r (random n))

123: (v (vector-ref vec r)))

124: (vector-set! vec r (vector-ref vec (1- n)))

125: (loop (1- n) (cons v ls1)))))))

126:

127: ;;; making a list of four-digit numbers in which numeral 0--9 appear once

128: (define (make-numbers)

129: (let ((ls1 '()))

130: (letrec ((rec (lambda (i num ls)

131: (if (= i 4)

132: (set! ls1 (cons (ls2nvec ls) ls1))

133: (for-each

134: (lambda (n)

135: (rec (1+ i) (delv n num) (cons n ls)))

136: num)))))

137: (rec 0 '(0 1 2 3 4 5 6 7 8 9) '()))

138: ls1))

139:

140: ;;;

141: (define (game-over sc0 sc1)

142: (write-to-stdout

143: (cond

144: ((= sc0 sc1) "Draw")

145: ((> sc0 sc1) "I won.")

146: (else "You won.")))

147: 'game-over)

148:

149: (define (scoring-user-guess an0 gu1)

150: (let ((sc1 (scoring an0 gu1)))

151: (show-user-score sc1)

152: sc1))

153:

154: ;;; Practical main function. tail recursive.

155: (define (mastermind-rec an0 candidates)

156: (if (null? candidates)

157: (error "Error. You gave wrong score for my guess, probably.")

158: (let ((gu0 (car candidates)))

159: (let ((sc1 (scoring-user-guess an0 (read-user-guess)))

160: (sc0 (read-my-score gu0)))

161: (if (or (= sc0 20) (= sc1 20))

162: (game-over sc0 sc1)

163: (mastermind-rec an0

164: (keep-matching-items

165: (cdr candidates)

166: (lambda (x) (= (scoring gu0 x) sc0)))))))))

167:

168: ;;; The main function called from the top-level

169: (define (mastermind)

170: (let ((ls0 (make-numbers)))

171: (mastermind-rec (list-ref ls0 (random (length ls0))) (shuffle-numbers ls0))))

|  |  |  |
| --- | --- | --- |
| **Functions** | **Comments** | **Lines** |
| **(1- x)** | It decrements **x** | 27 |
| **(char2int c)** | It converts a character **c** (#\0 --#\9) to an integer (0--9). | 30 |
| **(ls2nvec ls)** | It converts a list of 4 numerics (**ls**) to the vector expression. '(5 3 6 0) &rarr #(1 0 0 3 0 4 2 0 0 0) | 34 |
| **(nvec2intvec)** | It convers a vector expression **vec** to an ordinally integer. | 44 |
| **(int2str i)** | It converts a four digit integer **i** to a string. If **i** is less than 1000, '0' is inserted at higher positions. | 54 |
| **(read-from-stdin str)** | It displays **str** to the standard output and returns the line that user inputs from the standard input. | 64 |
| **(write-to-stdout . ls)** | It outputs each item of **ls** to the standard output and insert a line feed at the end. | 70 |
| **(str2nvecstr)** | It converts the user input string **str** for four digit number to the vector expression. | 75 |
| **(scoringvec0 vec1)** | It calculates the similarity of two integers (vector expression) **vec0** and **vec1** by (5\*Nbull + Ncow). | 85 |
| **(show-user-score score)** | It calculates Nbull and Ncow from the similarity **score** and display them to the standard output. | 98 |
| **(read-my-score gu0)** | It displays the guess by computer (**gu0**), let user input the Nbull and Ncow, and return the similarity score. | 105 |
| **(read-user-guess)** | It returns the vector expression of the guess of user. | 112 |
| **(shuffle-numbers ls0)** | It shuffles **ls0**. As random access is required to shulle, it converts the **ls0** to a vector and picks the items rondomly to produce a shuffled list. | 116 |
| **(make-numbers)** | It returns a list consisting of all four digit numbers of different numerics. | 128 |
| **(game-oversc0 sc1)** | It determines the winner by comparing the score of the computer (**sc0**) and the user (**sc1**). | 141 |
| **(scoring-user-guessan0 gu1)** | It calculates the similarity of the code of computer (**an0**) and the guess of the user (**gu1**) and outputs the Nbull and Ncow using show-user-score. | 149 |
| **(mastermind-rec an0candidates)** | The practical main function, it takes two arguments; the code of the computer (**an0**) and the list of guesses (**candidates**). It calculates the score of the computer (**sc0**) and the user (**sc1**) and calls (game-over **sc0** **sc1**) if **sc0** or **sc1** is 20. if not, it filters the candidates according to **sc0** (lines 164 – 166) and continues the game. | 155 |
| **(mastermind)** | Call this function from the console to start the game. | 169 |

**4.4. How to play**

Input following to start the game. It is better to compile befor play (You need to compile once.). Even the program is simple, it is difficult to win.

(compile-file "mastermind.scm")

(load "mastermind")

(mastermind)

**5. Summary**

In this chapter, I have explained about vectors and structures and played by mastermind. [The source code of the mastermind](http://www.shido.info/lisp/scheme_vec.zip) is attached.

I will explain about defining your own syntax in the next chapter. Ability of user defining syntax is one of the benefits of Lisp/Scheme.

**Answer of Exercise**

**Answer 1**

01: (define (inner-product vec1 vec2)

02: (let ((len1 (vector-length vec1))

03: (len2 (vector-length vec2)))

04: (if (= len1 len2)

05: (let loop ((i 0) (pro 0))

06: (if (= i len1)

07: pro

08: (loop (1+ i) (+ pro

09: (\* (vector-ref vec1 i)

10: (vector-ref vec2 i))))))

11: (error "different dimensions."))))