**15. Defining Syntax**

**1. Introduction**

In this chapter, I am going to explain how to define your own syntax. User define syntax is called macro. The macro of the Lisp/Scheme is much more powerful than that of other languages such as the C. Macro makes your program beautiful and tight.

**Macro is a transformation of codes.** Codes are transformed before being evaluated or compiled, and the procedure continues as if the transformed codes are written from the beginning.

In the Scheme, simple macros can be defined easily by using **syntax-rules** which is defined in the R5RS, while macro definition of the Common Lisp is complicated. By using the syntax-rules, you can define macros in a direct way without worrying about variable captures. On the other hand, defining complicated macros that cannot be defined using the syntax-rules is more difficult than that of the Common Lisp.

**2. Examples of Simple Macros**

I will show simple macros as examples.

[code 1] shows a macro that assigns '() to a variable.

[code 1]

(define-syntax nil!

(syntax-rules ()

((\_ x)

(set! x '()))))

The second argument of the syntax-rules is a list of the pair of the expressions before and after the transformation. The \_ represents the name of the macro. In short, [code 1] means that the expression **(nil! x)** is transformed into **(set! x '())**.

Such kind of procedures cannot be written by functions, because functions cannot affect the variable outside due to the closure. Let's write the function version of the [code 1] and see what happens.

[code 1']

(define (f-nil! x)

(set! x '()))

(define a 1)

;Value: a

(f-nil! a)

;Value: 1

a

;Value: 1 ; the value of a dose not change

(nil! a)

;Value: 1

a

;Value: () ; a becomes '()

I will show another simple example. Let's write a macro when, in which several expressions are evaluated when the predicate is true.

[code 2]

(define-syntax when

(syntax-rules ()

((\_ pred b1 ...)

(if pred (begin b1 ...)))))

The **...** in the [code 2] represents arbitrary number of expressions (including 0). [code 2] indicates that the expression

(when pred

b1

...)

is transformed to

(if pred

(begin

b1

...))

. This macro also cannot be written by a function because this is transformed into the special form if. Following shows how to use the when.

(let ((i 0))

(when (= i 0)

(display "i == 0")

(newline)))

i == 0

;Unspecified return value

I will show two practical macros; while and for. The while evaluates the body while the predicate is true. The for evaluates the body if the number is in the range.

[code 3]

(define-syntax while

(syntax-rules ()

((\_ pred b1 ...)

(let loop () (when pred b1 ... (loop))))))

(define-syntax for

(syntax-rules ()

((\_ (i from to) b1 ...)

(let loop((i from))

(when (< i to)

b1 ...

(loop (1+ i)))))))

Following shows how to use them.

(let ((i 0))

(while (< i 10)

(display i)

(display #\Space)

(set! i (+ i 1))))

0 1 2 3 4 5 6 7 8 9

;Unspecified return value

(for (i 0 10)

(display i)

(display #\Space))

0 1 2 3 4 5 6 7 8 9

;Unspecified return value

**Exercise 1**

Write a macro in that several expressions are evaluated when the predicate is false. (it is the opposite of the when.)

**3. More about the**syntax-rules

**3.1. Defining several patterns**

The syntax-rules can define several patterns. For instance, the macro that increments the value of a variable, **incf** increments the value of the variable by one if only the variable name is given, and by value if variable name and the value are given. The macro incf can be defined by writing several transformation patterns like [code 4].

[code 4]

(define-syntax incf

(syntax-rules ()

((\_ x) (begin (set! x (+ x 1)) x))

((\_ x i) (begin (set! x (+ x i)) x))))

(let ((i 0) (j 0))

(incf i)

(incf j 3)

(display (list 'i '= i))

(newline)

(display (list 'j '= j)))

(i = 1)

(j = 3)

;Unspecified return value

**Exercise 2**

Write a macro decf that subtracts a value from the variable. If the decrement value is omitted, it subtracts one form the variable.

**Exercise 3**

Improve the for shown in the [code 3] to accept a step width. If the step width is omitted, it is 1.

**3.2. Recursive definition of macros**

Forms or and and are macros and defined recursively like as follows.

[code 5]

(define-syntax my-and

(syntax-rules ()

((\_) #t)

((\_ e) e)

((\_ e1 e2 ...)

(if e1

(my-and e2 ...)

#f))))

(define-syntax my-or

(syntax-rules ()

((\_) #f)

((\_ e) e)

((\_ e1 e2 ...)

(let ((t e1))

(if t t (my-or e2 ...))))))

Some of complicated macros can be defined using recursive definitions.

**Exercise 4**

Define let\* by yourself.

**3.3. Using reserved words**

The first argument of the syntax-rules is a list of reserved words. For instance, cond is defined like [code 6], in which else is a reserved word.

[code 6]

(define-syntax my-cond

(syntax-rules (else)

((\_ (else e1 ...))

(begin e1 ...))

((\_ (e1 e2 ...))

(when e1 e2 ...))

((\_ (e1 e2 ...) c1 ...)

(if e1

(begin e2 ...)

(cond c1 ...)))))

**4. local syntax**

Local syntax can be defined using **let-syntax** and **letrec-syntax** in the Scheme. The usage of these forms is similar to that of the define-syntax.

**5. Implementation Depending Macro Definition**

The syntax-rules cannot define some kinds of macros. The way of defining such macros is prepared in Scheme implementations. You can skip this section as it is heavily depend on the implementations.

In the case of the MIT-Scheme, The **sc-macro-transformer** is available for such purpose, which allows to write macros in a similar way to taht of the Common Lisp. See [the Common Lisp HyperSpec.](http://www.lispworks.com/documentation/HyperSpec/Body/02_df.htm) about what **` , ,@** are. See MIT-Scheme manual about sc-macro-transfomrer and make-syntactic-closure. [code 7] shows simple examples.

[code 7]

(define-syntax show-vars

(sc-macro-transformer

(lambda (exp env)

(let ((vars (cdr exp)))

`(begin

(display

(list

,@(map (lambda(v)

(let ((w (make-syntactic-closure env '() v)))

`(list ',w ,w)))

vars)))

(newline))))))

(define-syntax random-choice

(sc-macro-transformer

(lambda (exp env)

(let ((i -1))

`(case (random ,(length (cdr exp)))

,@(map (lambda (x)

`((,(incf i)) ,(make-syntactic-closure env '() x)))

(cdr exp)))))))

(define-syntax aif

(sc-macro-transformer

(lambda (exp env)

(let ((test (make-syntactic-closure env '(it) (second exp)))

(cthen (make-syntactic-closure env '(it) (third exp)))

(celse (if (pair? (cdddr exp))

(make-syntactic-closure env '(it) (fourth exp))

#f)))

`(let ((it ,test))

(if it ,cthen ,celse))))))

The first macro **show-vars** is to show values of variables.

(let ((i 1) (j 3) (k 7))

(show-vars i j k))

((i 1) (j 3) (k 7))

;Unspecified return value

The form (show-vars i j k) is expanded like as follows. As macro can return only one expression, The begin is required to return sets of expressions.

(begin

(display

(list

(list 'i i) (list 'j j) (list 'k k)))

(newline))

The second macro **random-choice** is to select a value or procedure randomly from the arguments.

(define (turn-right) 'right)

(define (turn-left) 'left)

(define (go-ahead) 'straight)

(define (stop) 'stop)

(random-choice (turn-right) (turn-left) (go-ahead) (stop))

;Value: right

The form is expanded like as follows:

(case (random 4)

((0) (turn-right))

((1) (turn-left))

((2) (go-ahead))

((3) (stop)))

The third macro **aif** is an anaphoric macro. The result of the predicate can be refereed as **it**. The variable it is captured by letting the second argument ofmake-syntactic-closure be '(it).

(let ((i 4))

(aif (memv i '(2 4 6 8))

(car it)))

;Value: 4

Following shows the expansion.

(let ((it (memv i '(2 4 6 8))))

(if it

(car it)

#f))

**6. A Primitive Implementation of the Structure**

The structure can be implemented by a simple macro shown in [code 8]. The reality of the structure defined here is a vector and accessors and setters functions are created automatically by the macro. If your favorite Scheme implementation does not have structures, you should implement them by yourself.

[code 8]

01: ;;; simple structure definition

02:

03: ;;; lists of symbols -> string

04: (define (append-symbol . ls)

05: (let loop ((ls (cdr ls)) (str (symbol->string (car ls))))

06: (if (null? ls)

07: str

08: (loop (cdr ls) (string-append str "-" (symbol->string (car ls)))))))

09:

10: ;;; obj -> ls -> integer

11: ;;; returns position of obj in ls

12: (define (position obj ls)

13: (letrec ((iter (lambda (i ls)

14: (cond

15: ((null? ls) #f)

16: ((eq? obj (car ls)) i)

17: (else (iter (1+ i) (cdr ls)))))))

18: (iter 0 ls)))

19:

20:

21: ;;; list -> integer -> list

22: ;;; enumerate list items

23: (define (slot-enumerate ls i)

24: (if (null? ls)

25: '()

26: (cons `((,(car ls)) ,i) (slot-enumerate (cdr ls) (1+ i)))))

27:

28: ;;; define simple structure

29: (define-syntax defstruct

30: (sc-macro-transformer

31: (lambda (exp env)

32: (let ((struct (second exp))

33: (slots (map (lambda (x) (if (pair? x) (car x) x)) (cddr exp)))

34: (veclen (- (length exp) 1)))

35:

36: `(begin

37: (define ,(string->symbol (append-symbol 'make struct)) ; making instance

38: (lambda ls

39: (let ((vec (vector ',struct ,@(map (lambda (x) (if (pair? x) (second x) #f)) (cddr exp)))))

40: (let loop ((ls ls))

41: (if (null? ls)

42: vec

43: (begin

44: (vector-set! vec (case (first ls) ,@(slot-enumerate slots 1)) (second ls))

45: (loop (cddr ls))))))))

46:

47: (define ,(string->symbol (string-append (symbol->string struct) "?")) ; predicate

48: (lambda (obj)

49: (and

50: (vector? obj)

51: (eq? (vector-ref obj 0) ',struct))))

52:

53: ,@(map

54: (lambda (slot)

55: (let ((p (1+ (position slot slots))))

56: `(begin

57: (define ,(string->symbol (append-symbol struct slot)) ; accessors

58: (lambda (vec)

59: (vector-ref vec ,p)))

60:

61: (define-syntax ,(string->symbol ; modifier

62: (string-append

63: (append-symbol 'set struct slot) "!"))

64: (syntax-rules ()

65: ((\_ s v) (vector-set! s ,p v)))))))

66: slots)

67:

68: (define ,(string->symbol (append-symbol 'copy struct)) ; copier

69: (lambda (vec)

70: (let ((vec1 (make-vector ,veclen)))

71: (let loop ((i 0))

72: (if (= i ,veclen)

73: vec1

74: (begin

75: (vector-set! vec1 i (vector-ref vec i))

76: (loop (1+ i)))))))))))))

Following shows the usage:  
You can define a structure by giving slot names only or slot names and default values.

;;; Defining a structure point having 3 slots whose defaults are 0.0.

(defstruct point (x 0.0) (y 0.0) (z 0.0))

;Unspecified return value

(define p1 (make-point 'x 10 'y 20 'z 30))

;Value: p1

(point? p1)

;Value: #t

(point-x p1)

;Value: 10

;;; Default values are used for unspecified values when an instance is made.

(define p2 (make-point 'z 20))

;Value: p2

(point-x p2)

;Value: 0.

(point-z p2)

;Value: 20

;;; Changing a slot value

(set-point-y! p2 12)

;Unspecified return value

;;; The reality of the structure definde by [code 8] is a vector

p2

;Value 14: #(point 0. 12 20)

;;; Defining a structure 'book' with no default values.

(defstruct book title authors publisher year isbn)

;Unspecified return value

(define mon-month

(make-book 'title

"The Mythical Man-Month: Essays on Software Engineering"

'authors

"F.Brooks"

'publisher

"Addison-Wesley"

'year

1995

'isbn

0201835959))

;Value: mon-month

mon-month

;Value 15: #(book

"The Mythical Man-Month: Essays on Software Engineering"

"F.Brooks"

"Addison-Wesley"

1995

201835959)

(book-title mon-month)

;Value 13: "The Mythical Man-Month: Essays on Software Engineering"

**7. Summary**

I have explained about the macro in the Scheme briefly. Macros make your programs elegant.

The **syntax-rules** makes it easy to write macros. Writing macros of the Common Lisp, on the other hand, requires certain skill.

**Answers for the Exercises**

**Answer 1**

(define-syntax unless

(syntax-rules ()

((\_ pred b1 ...)

(if (not pred)

(begin

b1 ...)))))

**Answer 2**

(define-syntax decf

(syntax-rules ()

((\_ x) (begin (set! x (- x 1)) x))

((\_ x i) (begin (set! x (- x i)) x))))

**Answer 3**

(define-syntax for

(syntax-rules ()

((\_ (i from to) b1 ...)

(let loop((i from))

(when (< i to)

b1 ...

(loop (1+ i)))))

((\_ (i from to step) b1 ...)

(let loop ((i from))

(when (< i to)

b1 ...

(loop (+ i step)))))))

**Answer 4**

(define-syntax my-let\*

(syntax-rules ()

((\_ ((p v)) b ...)

(let ((p v)) b ...))

((\_ ((p1 v1) (p2 v2) ...) b ...)

(let ((p1 v1))

(my-let\* ((p2 v2) ...)

b ...)))))