Bulletin of the Educational Research and Development, Faculty of Education, Kagoshima University
2016, Special Issue No.6, 153-163

論 文

Scheme as yet another educational programming language

磯 川 幸 直[1]［鹿児島大学教育学系（数学教育）］
奥 平 敦[2]［鹿児島国際大学経済学部］

ISOKAWA Yukinao・OKUDAIRA Atuya

キーワード：Scheme、programmng language education、Bigloo、macro、Komachi problem

Abstract：Programming language Scheme is still used in education. But its share is not large. We used Scheme in our course of programming language education. In order to illustrate power of Scheme that can solve a complicated problem, a solution of Komachi problem is presented. Our conclusion is that Scheme is still a powerful language and fit for programming language education.

Key words：Scheme, macro, programming language education, Komachi problem

1 Introduction

At top Universities of the USA, Python is the most popular educational programming language; Other languages are Java, Matlab, C, C++, Scheme and Scratch [5]. Scheme is still used in education, but the population is smaller than that of C or Python. Although there are some critical opinions about Scheme [2], there are opinions that it is a very good language[4].

In this paper the authors propose several materials and tips for teaching computer programming by Scheme.

Scheme is a very simple language [1, 13, 12], much easier to implement than many other languages of comparable expressive power. This ease is attributable to the use of lambda calculus to derive much of the syntax of the language from more primitive forms. For instance, among 23 syntactic constructs defined in the RRS Scheme standard [3], 11 ones are written as macros involving more fundamental forms, principally lambda. As RRS says (RRS sec. 3.1):

The most fundamental of the variable binding constructs is the lambda expression, because all other variable binding constructs can be explained in terms of lambda expressions.

In 1998 Sussman and Steele, who are founders of Scheme, remarked that the minimalism of Scheme was not a conscious design goal, but rather the unintended outcome of the design process [14].

We were actually trying to build something complicated and discovered, serendipitously, that we

1 isokawa@edu.kagoshima-u.ac.jp
2 okudaira@eco.iuk.ac.jp

3 define, lambda, if, quote, unquote, unquote-splicing, quasiquote, define-syntax, let-syntax, letrec-syntax, syntax-rules, set!, do, let, let*, letrec, cond, case, and, or, begin, named let, delay

had accidentally designed something that met all our goals but was much simpler than we had intended....we realized that the lambda calculusâ €” a small, simple formalismâ €” could serve as the core of a powerful and expressive programming language.

RRS tried to depart from the minimalism and as a result it has some incompatibilities with RRS. RRS tries to harmonize RRS with RRS. Scheme has a lot of implementations. It may have a bad effect on keeping the share in languages [9]. Most implementations support RRS. Users have to check which RRS their Scheme supports.

Returning to education of Scheme, we are firmly convinced that the most important thing is to teach only fundamental principles with moderate complicated examples. On the other hand, it seems to be harmful to each materials of practical and/or academic nature by too simple or too complex examples. Fortunately, the core of Scheme is small. All principles that every beginner needs are concepts of function, recursion, higher-order function, list, macro and so on. Furthermore, the computational model of any Scheme program is a kind of disguise of secondary school algebra.

In the section 2, we describe the our schemes in education. In the section 3, to demonstrate large power of Scheme, we treat an arithmetric problem, which can be easily understood even by primary school pupils, but is difficult to solve both by “human power” and by traditional computer language such as C.

2 Schemes in education

We used Scheme in the courses of the programming language. Operating systems are Windows 7 and UNIX/Linux. We describe the way how it is used.

2.1 Development environment

In IUK to which the second author university belongs, we used a slightly modified mathlibre-debian [7] for the development environment. It can be used on a DVD, on a USB flash memory stick or on a Oracle Virtual Box. In the first day of class, we use mathlibre on DVDs. In the second day of class, the students installs mathlibre on their USB flash memory sticks. Using the sticks, the students can treat the PCs as if their own workstations. The default window system of mathlibre-debian is LXDE. it is easy to use for the students because LXDE has the Windows 7-like user interface.

We used the basic UNIX/Linux tools; terminal emulators, emacs and command line utilities. We used the modified version of emacs which has the menu items written in Japanese [8] because the students’ ability of reading english is limited.

As the scheme system, we use Bigloo [11]. Bigloo is a Scheme compiler which makes efficient stand alone binary executive files. It compiles to a Scheme code to a C code, and then a C compiler (gcc in case of Linux, Mingw in case of Microsoft Window) compiles the C code to an executable file. Furthermore, since Bigloo has an interface with C to import and export data, it can use full power of C to overcome some weakness of a very higher order language such as Scheme.

On the other hand, turning our eye to programming education, we can also use Bigloo as an interpretor. Thus beginners are released from a troublesome work of “compile”.

The original mathlibre-debian does not have Bigloo, (but it has an RRS Scheme script interpreter Gauche,) so we modified it so that it includes bigloo. One can let the students to install it from the tar.gz file. It may be educational, but we did not do so mainly because of time.

Similarly the first author uses an environment of Emacs and Bigloo in the courses of Kagoshima University, however in Windows 7 instead of Linux. This environment is installed into USB stick, and it is started-up by a batch file in the below (Here we assume that the drive E designates the place of USB stick).

```
echo off
rem intialization
set USR_INPUT_STR=
set /P USR_INPUT_STR=”Please input username:“
set str1=C:\Users\
set str2=\AppData\Roaming\.emacs.d
set dir1=%str1%%USR_INPUT_STR%%str2%
set str3=\AppData\Roaming\.emacs.d\init.el
set dir2=%str1%%USR_INPUT_STR%%str3%
rem mkdir C:\Users\username\AppData\Roaming\.emacs.d
mkdir %dir1%
copy E:\emacs-24.3\init.el %dir2%
E:\emacs-24.3\bin\runemacs.exe
```

2.2 Cource contents

In IUK, Scheme is used in the one semester course named programming II. It is an essential part of the curriculum for the licence of highschool teachers who teach the subject area Information. It assumes little experience of programming. Although the limitation of the time forced us to select the topics, we selected not only standard cons-pair list manipulations but also vectors. Students can compare the insertion sort of list and vector. The performance of them is comparable with each other.

Similarly the first author teaches Scheme in two classes: in one class students have no experience of programming, but in another class they have one-year experience of C programming. He presents common materials to both classes, although additional explanations about C codes in the latter class. These materials consist of fundamental concepts such as recursion, higher-order function, and macro. One material will be discussed in the section 3.

3 Komachi problem

In this section, by explaining how Scheme can be used to solve a complicated problem, we show various

advantages that we can gain by using Scheme in programming education. As an example of complicated problem, we consider the Komachi problem.

Th Komachi problem is as follows: Using the digits 1 through 9 in consecutive ascending order and any mathematical operations including the four basic ones +, − , × , ÷ , make the formula equal to 100.

The Komachi problem was studied a long time ago [3], but it attracts us even in modern ages [10]. In [3] the problem using only +, − was studied by man-power, and in [10] the problem using all four basic ones +, − , × , ÷ was studied by making a Fortran program. In this section we consider the problem not only all four basic ones +, − , × , ÷ but also "parentheses" .

3.1 A macro that implements "list comprehension"

If we use only simple "loop" syntax, as loop be nested many times, our program becomes so complex that we can not understand it. In such a multi-loop situaton, "list comprehension" is a syntax sugar. Although Scheme does not provide this syntax, we can easily furnish an alternative way using macro. The following is an example.

```
(define-syntax derive-list
  (syntax-rules (<- ! let)
    ((_ e) e)
    ((_ (n <- g) (! p) rest ...)
      (deriv-list (n <- (filter (lambda (n) p) g)) rest ...))
    ((_ (n <- g) (list expr))
      (map (lambda (n) expr) g))
    ((_ (n <- e) rest ...)
      (apply append (map (lambda (n) (derive-list rest ...)) e)))
    ((_ (let bindings) rest ...)
      (let bindings (derive-list rest ...)))
    ((_ (! p) rest ...)
      (if p (derive-list rest ...) '() ))
   ))
```

By use of the macro we can define the final function solutions succinctly.

```
(define (solutions ns n)
  (derive-list
    (ms <- (paste ns))
    (r <- (exprs ms))
    (! (= (cdr r) n))
    (list (car r))))
```

The content of the above code can be expanded into double loop as follows:

```
(apply append (map (lambda (ms)
  (apply append (map (lambda (r)
       (if  (= (cdr r) n)
         (list (car r))))
       )
     (exprs ms))))
   (paste ns))))
   )
```

3.2 Paste digits

The following function paste make all combinations of pasted numbers from given digits. For example, we have

```
(paste '(1 2 3 4))
  => ((1 2 3 4) (1 2 34) (1 23 4) (1 234) (12 3 4) (12 34) (123 4) (1234))
```

In Scheme it is easy to implement this function, for example, as follows:

```
   (define (paste xs)
     (cond ((null? xs) '())
           ((null? (cdr xs)) (list xs))
           (else
             (append
               (map (lambda (y) (cons (car xs) y)) (paste (cdr xs)))
               (let* ((x1 (car xs)) (x2 (cadr xs)) (y (+ (* 10 x1) x2))
                 (ys (cons y (cddr xs))) (paste ys))
             ))
     ))
```

To make comparison, we write a C code that has the same structure as the Scheme code. In the following C code, variables xtemp, ytemp[M] are only used to store intermediate results temporarily. Scheme code becomes far more simple than C because Scheme could do well without temporary variables.

```
#define N 9
#define M 256

typedef struct {
  int n;
```

```
  int v[N];
} List;

int paste(List x, List y[])
{
  int m, mtemp;
  List xtemp, ytemp[M];
  int i, j;

  if (x.n == 0) { return 0; }
  if (x.n == 1) { y[0].n = 1; y[0].v[0] = x.v[0]; return 1; }

  xtemp.n = x.n - 1;
  for (i = 0; i <= N - 2; i++) xtemp.v[i] = x.v[i+1];
  mtemp = paste(xtemp, ytemp);
  for (j = 0; j < mtemp; j++) {
    y[j].n = ytemp[j].n + 1;
    y[j].v[0] = x.v[0];
    for (i = 1; i <= N - 1; i++) y[j].v[i] = ytemp[j].v[i-1];
  }
  m = mtemp;

  xtemp.n = x.n - 1;
  xtemp.v[0] = 10 * x.v[0] + x.v[1];
  for (i = 1; i <= N - 2; i++) xtemp.v[i] = x.v[i+1];
  mtemp = paste(xtemp, ytemp);
  for (j = 0; j < mtemp; j++) {
    y[m+j].n = ytemp[j].n;
    for (i = 0; i <= N - 1; i++) y[m+j].v[i] = ytemp[j].v[i];
  }
  m += mtemp;
  return m;
}
```

3.3 Construct numerical expressions

We begin by defining numerical expression (in other word, formula) recursively:

- a number is a numerical expression.
- if both `l,r` are numerical expressions, then `(+ l r), (- l r), (* l r), (quotient l r)` are also numerical expressions.

The following function `(exprs ns)` construct all "admissible" numerical expressions using a list of numbers `ns` (Exact meaning of "admissible" will be explained later). For example,

```
(exprs '(1 2 3)) =>
  ( (+ 1 (+ 2 3)) (- 1 (+ 2 3)) (+ 1 (- 2 3)) (+ 1 (* 2 3))
    (- 1 (* 2 3)) (* (+ 1 2) 3) (quotient (+ 1 2) 3) (* (- 1 2) 3) )
```

```
(define (exprs ns)
  (cond ((null? ns) '())
        ((null? (cdr ns)) (let ((n (car ns))) (list (cons n n))))
        (else (derive-list
                (lsrs <- (split ns))
                (l <- (exprs (car lsrs)))
                (r <- (exprs (cadr lsrs)))
                (combine l y)))))
```

```
(define (split xs)
  (cond ((null? xs) '())
        ((null? (cdr xs)) '())
        (else
          (let ((x (car xs)) (xs (cdr xs)))
            (cons (list (list x) xs)
              (map
                (lambda (zss)
                  (let ((ls (car zss)) (rs (cadr zss))) (list (cons x ls)
rs)))
                (split xs)
              )
            )))))
```

The function `(split ns)` splits a given into two parts. For example,

```
(split '(1 2 3 4)) => (((1) (2 3 4)) ((1 2) (3 4)) ((1 2 3) (4)))
```

The function (combine l r) constructs at most four numerical expressions from a left numerical expression l and a right one r. It can be implemented simply as follows:

```
(define (combine l r)
  (let ((w '()))
    (if (admissible-add-subtract? l r)
      (begin
        (set! w (list (list '+ l r)))
        (set! w (append w (list (list '- l r))))))
    (if (admissible-multiply? l r)
      (set! w (append w (list (list '* l r)))))
    (if (admissible-divide? l r)
      (set! w (append w (list (list 'quotient l r)))))
    w
  ))
```

At last, we show what kind of numerical expressions admissible are. To explain these briefly, we will consider only `admissible-divide?`. Here, first we evaluate both left and right numerical expressions `l, r`. Results of evaluation are bound to variables x,y. We would like to divide of the left expression by the right one. Then, if the value y is equal to zero, the division causes "division by zero error". If x is not a multiple of y, the division results into a fraction. Thus the division is amissible only when `(and (not (= y 0)) (= (modulo x y) 0))` is true.

```
(define (admissible-add-subtract? l r)
  (if (pair? r)
    (if (pair? l)
      (let ((lop (car l)) (rop (car r)))
        (cond ((and (or (eq? lop '+) (eq? lop '-))
                    (or (eq? rop '+) (eq? rop '-))) #f)
              (else #t)))
      #t
    )
    (if (pair? l)
      #f
      #t
    )
  ))

(define (admissible-multiply? l r)
  (if (pair? r)
```

```
    (if (pair? l)
      (let ((lop (car l)) (rop (car r)))
        (cond ((and (eq? lop '*) (eq? rop '*)) #f)
              (else #t)))
      #t
    )
    (if (pair? l)
      #f
      #t
    )
  ))

(define (admissible-divide? l r)
  (let ((x (eval l)) (y (eval r)))
    (and (not (= y 0)) (= (modulo x y) 0))))
```

Numerical expressions are represented by lists, thus a kind of data. But, from another point of view, they are programs which we can evaluate (i.e. compute). In other words, while our Komachi code continues to construct numerical expressions one after another, it automatically write another code (or other codes). A direct consequence of this recognition is that we need not be always conscious of evaluation, because we can do evaluate expressions in our hand whenever as we like.

At last we report that there are 1441 solutions to the Komachi problem that admits use of "parentheses". The following are examples of solutions:

```
(* (quotient 12 3) (quotient (* 45 (+ 6 (- 7 8))) 9))
(quotient (* 12 (+ 34 (- 56 (+ 7 8)))) 9)
(+ 1234 (- 5 (* 67 (+ 8 9))))
(- (+ 1234 5) (* 67 (+ 8 9)))
```

4 Discussion

The defining characteristics of Scheme are a statically scope, latent typing, unlimited extent of objects, being properly tail recursive, procedures as objects, first-class continuations and evaluation of arguments before the procedure call. Many of the characteristics are shared with other languages.

Scheme is a multi-paradigm language. It supports procedural programming style, functional programming style and message-passing (object-oriented) style.

Scheme has an unusual expression i.e. `(+ 1 2)` for 1+2. This is a matter of taste. A Python function which

calculates Fibonacci number taken from a excellent textbook using Python [6] is as follows:

```
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

A Scheme procedure which calculates Fibonacci number is as follows:

```
(define (fib n)
  (if (or (= n 0)(= n 1))
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

Which do you like? Basically, the two codes resemble with each other. In Python, indentations of lines are essential and the end of the region of the function definition is implicit. The parenthesis of Scheme make the region of the expressions explicit. Modern editors like emacs support Python indentations and Scheme parenthesis checks. so you do not have to be afraid of them.

Performance is different between the languages and implementations. In printing fib(0) to fib(39) on a PC, Python 2.7.9 needs 39.800s, Python 3.4.1 needs 45.980s, Gauche 0.9.4 needs 14.916s and Bigloo 4.2c needs 3.024s when the code is compiled with -O6 option. Although performance may not be crucial to the education, it is a important characteristic.

The solution of Komachi problem described the previous section shows the power of Scheme. The solution would be more difficult in other languages like C.

Our conclusion is that programming education by Scheme is better than that by other languages such as C and Python because (1) Codes by Scheme are far shorter than those by C and thus intelligible. We can do without (i) intermediate temporarily variables, (ii) multi-loop, even (iii) simple loop, because (i) Scheme in principle does not use mutable variables, (ii) we can recourse to list comprehension via a macro, and (iii) Scheme can uses higher-order functions. Of course, Python has these advantages alike Scheme. However, (2) Codes by Scheme are more powerful than those by Python. For examples, numerical expressions in Scheme are data and programs simultaneously. Other languages than Scheme data and programs are different. Accordingly, if we make, for example, a program that solves the Komachi problem, we need carry out two mental processes in parallel: one is to construct data, another is to evaluate. These parallel thinking will result in more difficulty to implement any code to cope with more complicated problem. Returning to the Komachi problem ommiting use of “parentheses”, it seems to be beyond one intelligence to implement any code to solve the problem.

As we see yet, if students learn Scheme, they will gain power to write shorter programs for more difficult problems. However, if students learn less powerful languages, then would not write, or even if they could, they would gain far longer codes.

Acknowledgement

We thank the students of our lectures and the staff of the information processing center of the IUK. We also thank NAKAO Yasushi for useful discussions. A.O. expresses his gratitude to Manual Serrano and members of Indes team of Inria Sophia-Antipolis for their kind support during the sabbatical stay.

References

[1] Norman I Adams IV, David H Bartley, Gary Brooks, R Kent Dybvig, Daniel Paul Friedman, Robert Halstead, Chris Hanson, Christopher Thomas Haynes, Eugene Kohlbecker, Don Oxley, et al. Revised report on the algorithmic language scheme. ACM Sigplan Notices, 33(9):26–76, 1998.

[2] Richard Gabriel. The rise of â ∈ œworse is betterâ ∈ . Lisp: Good News, Bad News, How to Win Big, 2:5, 1991.

[3] Nakane Genjyun. Kanjya otogi zoshi. http://www.ndl.go.jp/math/e/s2/3.html, 1743.

[4] Paul Graham. Beating the averages, 2005.

[5] Philip Guo. Python is now the most popular introductory teaching language at top us universities. BLOG@ CACM, July, 2014.

[6] John V Guttag. Introduction to Computation and Programming Using Python. Mit Press, 2013.

[7] Tatsuyoshi Hamada. Mathlibre: Personalizable computer environment for mathematical research. ACM Commun. Comput. Algebra, 48(3/4):116–117, February 2015.

[8] Shinsuke IRIE. menu-tree.el. https://www.emacswiki.org/emacs/menu-tree.el, 2010.

[9] Atuya Okudaira. Will R7RS survive? The Kagoshima Journal of Economics, 51(4):425 – 446, 2011. in Japanese.

[10] Jr. Patton, Robert L. The 100 problem revisited. Journal of Recreational Mathematics, 4:276–280, 1971.

[11] Manuel Serrano and Pierre Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In Proceedings of the Second International Symposium on Static Analysis, pages 366–381. Springer-Verlag, 1995.

[12] Alex Shinn, John Cowan, Arthur A Gleckler, et al. Revised report on the algorithmic language scheme. 2013.

[13] Michael Sperber, R. Kent Dybvig, and Matthew Flatt. Revised [6] Report on the Algorithmic Language Scheme. Cambridge University Press, 2009.

[14] Gerald Jay Sussman and Guy L Steele. The first report on scheme revisited. Higher-Order and Symbolic Computation, 11(4):399–404, 1998.