福山大学工学部紀要
第20巻(1996)第1号

Scheme処理系の作成報告

今井光祐・河野俊彦*

Report on an Implementation of the Language Scheme

Mitsuhiro IMAI and Toshihiko KOUNO*

ABSTRACT

Our implementation of scheme almost satisfies the requirements in Revised^4 Report on the Algorithmic Language Scheme, that means tail recursions are properly treated and continuations are fully supported. This paper describes the language scheme briefly and explains the data structure for scheme objects, the implementation of environments for the static scope of variables, and the main structure of our scheme processor .

Keywords: Computer Language, Lisp, Scheme, Interpreter
キーワード：計算機言語、リスプ、スキーム、インタープリタ

1. まえがき

計算機言語は、単に計算機に計算の手順を指示するだけでなく、自然言語と同様、思考の手段を提供しなければならない。言語を思考の道具と捕らえるとき、皮肉にも、言語が思考を制限しているという側面に気付くことがある。Scheme[1, 2]は、G. J. Sussman と G. L. Steel Jr. によって設計されたLispの方言の1つであり、言語としてはもっとも小さいものの一つである。しかし、洗練された言語機能を厳選して提供することによって、言語が合わせ持つ制限を最小限に留めて、ユーザに多大な自由度を提供している。

今日、Common LispがLispの標準と見なされるのに対して、Schemeは教育用言語と見なされることが多い。文法や文の意味が非常に簡明なので、プログラミングの経験のない者でも、早期に抽象化の技法やアルゴリズムなどプログラミングの本質の修得に専念できるからであろう。しかし、汎用言語として見ても、Common Lispに多大な影響を与えたほどの優れた言語であり、ある面ではCommon Lisp以上に優れた能力を備えている。

筆者の一人（今井）は、簡明な文法・多様な能力に感銘を受け、いつの日か自らの手でScheme処理系を作ってみたいと考えていた。また、ガーベジコレクションや動的な環境の実現法などの実験の道具を手に入れる必要性を感じるようになってきた。言語としての魅力とともに、言語仕様の小ささは、このような実験の道具として最適のように思え、Scheme処理系を作成することにした。

すでに、市販の処理系もあり、フリーソフトウェアにも優れた処理系が存在する。まだ、それらに比べられるほどではないが、ひとまず、Schemeと呼べるものができたので、ここに報告する。

第2節では、簡単にSchemeの特徴を紹介する。第3節では、作成したScheme処理系（以後、aschと呼ぶことにする）の構造について述べる。第4節では、反省点、今後の課題について述べる。付録には、Schemeの計算能力の一端を示す。

2. Schemeの特徴

第1節でも述べたようにSchemeはLispの1つの方言であり、その特徴の多くは、他のLispと共通する。この節では、Schemeを他のLispから際だたせている特徴について述べる。

評価の一様性

Schemeでは、S式のすべての要素は同じ評価機構

*情報処理工学科

によって評価される。S式がリストである場合には、特殊形式（special form）と呼ばれる特別な場合を除いて、第一要素の評価値は手続きオブジェクトと見なされて他の要素の評価値に適用される。伝統的なLispが引数を評価するしないによって関数を2種類に分けていたのに比べて、この評価機構は非常にわかりやすく、特に、初心者にとってプログラミングを容易にしている。

静的な変数スコープ・手続きの第一級市民化

Schemeでは、他のAlgol系の言語のように、各変数はソースリストから明らかとなる静的なスコープを持つ。また、手続き（関数）は第一級市民として取り扱われる。すなわち、手続きオブジェクト（クロージャ）は、数値やリストといった他のオブジェクトとまったく同様に、変数に代入したり、手続きからの戻り値となることができる。手続きオブジェクトは、lambda形式を評価することによって生成され、評価時の環境と手続き本体を1つにまとめて保存したものである。保存された環境は、静的な変数スコープを実現するのに用いられる。すなわち、手続きオブジェクトが実際の引数に適用されるとき、仮引数に実引数を束縛したもの（lambda束縛）を保存されていた環境の上につけ加えて新しい環境を作り、その環境の上で手続きの本体が評価される。手続きの第一級市民化とこのような評価機構は、Schemeの言語としての記述能力を飛躍的に高める。その一端として、付録にSchemeとλ計算との親和性を示しておく。

継続の第一級市民化

ある時点以降になされるであろう計算を継続（continuation）[1, 3]と呼ばれる一つのオブジェクトとして取り扱うことができる。継続を正式にサポートした言語は、筆者らの知る限りSchemeだけと思われるので、図1に簡単な例を示して説明する。

doubleは初期値が空リストである変数として、また、receiverは、ある継続を受け取り、変数doubleに保存した後、値1を返す手続きとして定義されている。call/cc（call-with-current-continuationの省略形）は、継続を生成する組み込み関数であり、その呼び出し(call/cc receiver)は、継続（この例では、2を掛けてトップレベルに戻ること）を生成して、それをreceiverに渡す。receiverは1を返すので、(call/cc receiver)の評価値は1となるが、戻ってきた時点では、receiverの副作用によってdoubleには「2を掛けてトップレベルに戻る」という継続が保存されている。したがって、その後の(double 5)の評価では10を返し、(+ (double 5) 1)の評価では、1を加えるという操作は実行されず、(double 5)の評価後、10を返してトップレベルに戻る。

```
>(define double '())
double
>(define receiver
   (lambda (continuation)
     (set! double continuation)
     1))
receiver
>(* (call/cc receiver) 2)
2
>(double 5)
10
>(+ (double 5) 1)
10
```

Fig.1 An Example Using Continuation

この例のように継続は他のオブジェクトと同様に取り扱うことができて、非局所的な脱出やコルーチンなど、catch、throwなどに比べると、高度な実行の制御を可能としている。

末尾再帰の処理

通常、末尾再帰の処理は、最適化の一つとして処理されることが多いが、Schemeでは、仕様として、「末尾再帰は一定の記憶領域を用いる繰り返しとして処理されなければならない」と規定している。したがって、Schemeユーザは、効率について考慮することなく繰り返しを再帰として書けばよい。

末尾再帰の処理は、次のようにすれば、簡単に実現することができる。Schemeでは、lambda式の本体のように複数の式が書けるところでは、すべて、最後の式の評価値が複合式全体の評価値となる（暗黙のPROGNとして取り扱われる）。したがって、末尾再帰であるかどうかに関わらず、複合式の中の最後の式の評価に移るとき、CALLによる呼び出しでなくGOTOによるジャンプとすることができる。このようにすると、呼び出しに伴うオーバーヘッドがなくなる分、高速化することができる。同時に、末尾再帰も自動的に適切に処理されるようになる。

3. Scheme処理系aschの構造

実効速度を考えると不利であるが、どのマシンでも動作することを優先させた。そのために、解釈実行型の処理系（インタープリタ）とし、C言語で書いた。Scheme（Lisp）では、プログラムも他のデータと同じS式として表現され、それが読み込まれるとペア（セル）を用いた内部表現に変換される。内部表現への変換は、S式がプログラムである場合には、中間言語への変換と見なすことができる。たとえば、if 形式の処

理において偽のパートへジャンプしなければならないときでも、ポインタを2つ手繰るだけで済む。したがって、解釈実行型という言葉から受ける印象ほどの速度低下はない。ハードウェアの進歩を考慮すれば、可搬性を取る方が賢明であると言える。

さて、Scheme（Lisp）は、対話型の言語であり、READ-EVAL-PRINTループとよばれる「ユーザからS式を1つ読み込み、それを評価（計算）して、結果を印字する」という動作を繰り返す。以下に、aschで用いているデータ構造、環境とその表現法、READ-EVAL-PRINTループのそれぞれの部分でのaschが行っている処理を簡単に紹介する。

データ構造

aschでは、ほとんどすべてのオブジェクトの内部表現に図2のような単一のデータ構造を用いている。タグ部とユニオン部からなり、オブジェクトの実体はユニオン部に格納し、タグ部には実体が何であるかといった情報を格納する（ガーベジコレクションに用いるマークビットなどaschが内部的に用いる情報もタグ部に置いている）。これらは、最初に一括してメモリを割り当てた後、フリーリストと呼ぶリストの形にまとめておき、必要に応じて使用する。フリーリストに、十分なペア（セル）がなくなったときには、ガーベジコレクターを起動して、不要となったペアを回収する。aschでは、今の所、単純なマーク・スウィープ方式のガーベジコレクターを用いている。

```
struct sob {
  unsigned short tag;
  union {
    struct {long       bits;
            struct sob *rest;
            } integer;
    struct {struct sob *numerator;
            struct sob *denominator;
            } rational;
    double    real;
    struct {struct sob *realPart;
            struct sob *imaginaryPart;
            } complex;
    short     boolean;
    short     character;
    char      *string;
    struct {char       *name;
            struct sob *value;
            } symbol;
    struct {struct sob *car;
            struct sob *cdr;
            } pair;
    struct {unsigned   dim;
            struct sob **entries;
            } vector;
    struct {struct sob
                 *(*proc)(struct sob *);
            short      type;
            } primitive;
    struct {struct sob *definition;
            struct sob *environment;
            } compound;
    struct {struct sob *result;
            struct sob *proc;
            } promise;
    struct {FILE       *filePtr;
            struct sob *fileName;
            } port;
  } object;
};
```

Fig.2 The data structure for scheme objects (continued)

Fig.2 The data structure for scheme objects

環境とその表現

ここで言う環境とは、名前と実体との対応（変数束縛の集合）のことである。通常、Schemeで変数を束縛するには、let形式、letrec形式などの特殊形式を用いることが多い。しかし、これらの束縛形式は、手続きオブジェクトの適用時に行われるlambda束縛として実現することができる。そこで、aschでは、let形式やletrec形式はマクロとして取り扱い、最初の評価時にlamda形式に変換するようにしている。このようにlambda束縛だけに限定することによって、環境を操作する場所を限定することができる。

さて、現在、aschでは、lambda束縛を表現するフレームとして、仮引数（名前）のリストと実引数（実体）のリストのペアを用いている。そして、フレームのリストとして環境を表現している。このような環境の表現法は、直感的でわかりやすい反面、ある名前に対する束縛値を取り出す場合にはリストを順番に手繰っていく必要がある。最悪の場合、環境に含まれる名前の個数Nに比例する時間が掛かってしまう。そこで、aschが用意するグローバルな環境はハッシュ表にして高速化をはかっている。

READ部

この部分では、ユーザが入力した文字列（S式）を内部表現に変換する。大部分は、Schemeの組み込み手続きreadと共用している。翻訳系の教科書などでは、字句の切り出し部分、構文を解析する部分に分けられていることが多い。しかし、S式の構文は非常に簡単なので、aschでは、字句の切り出し部を再帰下降型の構文解析部に組み込んでいる。そして、字句の切り出し・構文解析を同時に行いながら、解析できた

部分から、逐次、内部表現に変換している。名前を読み込んだときには、それが特殊形式（special form － if 形式などのように 評価の一様性規則に従わない形式）のキーワードであるかどうかの判断などもこの部分で行っている。しかし、特殊形式として正しい形式であるかどうかのチェックは、次のEVAL部で行うようにしている。

EVAL部

この部分では、内部表現されたＳ式をある環境の上で評価する。評価は再帰的なので、再帰的に記述できそうだが、末尾再帰を正しく処理できなくなってしまう。そこで、Abelson 達のExplicit-Contorol Evaluator [4]を参考に、２本のスタックを用意して、評価を行っている。大まかに評価の様子を示すと次のようになる（再帰的に説明するが、実際には、用意したスタックと goto 文を用いて 再帰的な動作を明示的に記述している）。

1. まず、Ｓ式のタグ部を見て、自己評価的であるかどうか、すなわち、数値や文字列などのように評価した値が自分自身であるかどうか調べる。
2. 自己評価的であれば、何もせずにＳ式をそのまま評価値として返す。
3. 自己評価的でなければ、タグ部を見て名前かどうか調べる。
4. 名前であれば、現在の環境の中から名前に束縛されている値を評価値として返す。
5. 名前でなければ、Ｓ式最初の要素のタグ部を調べて、 lambda 形式や if 形式のような特殊形式 であるかどうか調べる。
6. 特殊形式であれば、それぞれの特殊形式に応じた評価を行う。特殊形式の多くはマクロとしてあるので、必要に応じて、より基本的な特殊形式を用いた形に変換した後、５へ戻って評価し直す。一度マクロ展開したものは保存しておき、二度手間を省くようにしている。
7. 特殊形式でなければ、手続きの適用であると見なして、手続き部、引数部を評価した後、実引数に手続きを適用する。
8. 手続きがユーザ定義の手続きであれば、手続きオブジェクトの一部として保存されている環境の上に仮引数に実引数を束縛したフレームをつけ加えて新しい環境を作り、その上で手続き本体を構成するＳ式を順番に評価していく。手続き本体の最後のＳ式へ GOTO でジャンプする。
9. 手続きが組み込み手続きであれば、対応するＣ言語で記述した関数を呼び出す。しかし、組み込み手続きの内、call/cc や delay、force のように制御構造に関係した手続きなどは別扱いして、それぞれに応じた処理を行っている。

PRINT部

この部分では、評価の結果である内部表現をユーザに理解できる外部表現に変換する。大部分は、組み込み手続き print、display と共用している。整数（多倍長整数）の変換など、手間取る部分もあるが、再帰的に記述してあり、READ-EVAL-PRINT ループの中ではもっとも簡単な部分である。

そのほか、asch は、初期化のための関数群、環境を操作するための関数群、記憶域の管理に関する関数群、組み込み手続きを記述した関数群、多倍長整数の演算に関する関数群などの部分からなる。

4. 今後の課題

asch は、もともと、ガーベジコレクションなどの実験用として作成し始めたものである。しかし、独立した Scheme 処理系としても有用なものとしたいと考えるようになった。現在、複素数のサポートと character-ready という手続きを除いて、ほぼ、言語の仕様を満足するものとなっている。多倍長整数、有理数まで扱えるようにしたことによって、数式処理系の研究や数論的な問題の実験的検証などにも用いることができるようになった。しかし、デバグ機能を持たないので、実用的な処理系と呼ぶには少し無理がある。また、インタープリタであるので、計算速度の面においても、不満が出てきた。

計算速度については、Ｓ式を読み込むと同時に、仮想的なマシン（たとえば、Ｐコードマシン）の命令コードに翻訳し、そのコードを解釈実行するような方式に変更しようかと考えている。このような方式は移植性に優れているという利点を持つが、不用意な翻訳は、計算速度の向上が望めないばかりか、かえって、速度低下を招く。現在、Scheme 処理系にとってどのような命令体系が優れているか研究中である。デバグ機能は、この命令体系の中にデバグ機能を容易にするための命令を組み込んでおくことによって実現しようと考えている。

Scheme の仕様では、ユーザマクロ機能については全く規定していない。しかし、ユーザマクロ機能は言語の柔軟性を飛躍的に高め、ヒープのダンプ機能を付加すれば、Scheme をエンジンとする多様な処理系へと発展する可能性を持つ。したがって、ぜひ、実現し

たいと考えている。

謝辞

広島大学工学部　伊藤雅明助教授には、バグの発見と共に、たえず、励ましていただいきました。管理工学研究所麻布分室　佐藤茂氏には、バグとその原因の発見に協力していただきました。両氏には、心から感謝の意を表明いたします。

付録　Scheme とλ計算

Scheme では、関数オブジェクトを第一級市民と認めることによって、Lisp の他の方言に比べて比較的簡単にλ計算を実現することができる。言い換えれば、Scheme はλ計算にきわめて近い計算機構を備えていると言える。2つの例を用いてこのことを示す。

例1　church の数

後藤は、λ計算をLisp 上で実現しようと試みた悪戦苦闘の様子を示して、Lispの 計算機構がλ計算のそれと同じでないことを示している[5]。同じ例が Scheme では、いとも簡単に実現できることを示そう。

λ計算において、数を表現するには、たとえば、次のように対応づければよい。

```
0 ==> λfx.x
1 ==> λfx.fx                                    (a-1)
2 ==> λfx.f(fx)
3 ==> λfx.f(f(fx))
...
```

上のように表現された数は church の数と呼ばれている。このとき、successor function （ある数よりも1大きい数を返す関数）は、次式で与えられる。

```
λabc.b(abc)                                     (a-2)
```

さて、Scheme 上でλ項(a-2) が successor function であることを確かめて見よう。λ抽象が右結合的であり、適用が左結合的であることに注意して、church 数2と successor function をその定義通りに Scheme で書いてみると次のようになる。

```
(define two
  (lambda (f)
    (lambda (x) (f (f x)))))

(define successor
  (lambda (a)
    (lambda (b)
      (lambda (c) (b ((a b) c))))))
```

さらに、次のように maybe-three を定義してみる。

```
(define maybe-three (successor two))
```

この maybe-three は two に successor を適用したものと定義したのだから、church 数3に等価なものとなっているはずである。すなわち、maybe-three は、関数として、(lambda (f) (lambda (x) (f (f (f x))))) と同じ振る舞いをするはずである。実際、

```
((maybe-three add1) 0)
```

と入力すれば、3という答えが返ってくる。0に +1 という操作を3回施したからである。また、

```
((maybe-three cdr)
 '(0 1 2 3 4 5 6 7 8 9))
```

と入力すれば、(3 4 5 6 7 8 9) という答えが返ってくる。(0 1 2 3 4 5 6 7 8 9) というリストに cdr（先頭の要素を除いたリストを値として返す関数）を適用するという操作を3回繰り返したからである。

上の結果から、maybe-three が church 数3に等価であること、したがって、(a-2) 式が successor function となっていることが理解できるであろう。

例2　不動点定理を用いた再帰的関数の定義

もう1つの例として、不動点定理、不動点オペレータを用いて再帰的な関数を定義してみよう。λ計算における不動点定理とは、次の事実のことである[6]。

任意のλ項 H に対して、HX = X をみたすλ項 X が存在する。

実際、不動点演算子の1つ であるY-コンビネータ

```
Y = λh.(λx.h(xx))(λx.h(xx))                     (a-3)
```

を用いると、X = YH が解となる。

さて、階乗関数 fac をλ項として表すと、次のようになる。

```
fac = λn.if(zero? n)
          1
          (* n (fac (sub1 n)))
```

したがって、階乗関数 fac は、Hを

```
H = λfn.if(zero? n)
          1
          (* n (f (sub1 n)))                    (a-4)
```

としたときのX＝HXの解である。(a-3)、(a-4)式をSchemeで表してみると、

```
(define y
  (lambda (h)
    ((lambda (x) (h (x x)))
     (lambda (x) (h (x x))))))

(define h
  (lambda (f)
    (lambda (n)
      (if (zero? n)
          1
          (* n (f (sub1 n)))))))
```

となるはずである。そして、facは(y h)と定義すればよさそうである。しかし、(define fac (y h))と入力すると、無限のループに陥ってしまう。理由は次のようなものである。yをhに適用すると、次のS式を評価しようとする。

```
((lambda (x) (h (x x)))
 (lambda (x) (h (x x))))          (a-5)
```

しかし、その評価を終えるためには、xに(lambda (x) (h (x x)))が束縛された環境下で下線部の(x x)というS式を評価しなければならず、全く同じ(a-5)式を評価しなければならないからである。

現実の計算機で無限を扱う1つの方法は、遅延評価を用いることである。すなわち、遅延手続きdelayを用いて、必要になるときまで(x x)の評価を遅らせればよい。delayで遅らせた評価は、forceを用いて評価を明示的に指示をしてやる必要がある。どれをforceするかは、(x x)がhの引数となっていることから、hの中のfであることがわかる。結局、y、h、facを次のように定義すればよい。

```
(define y
  (lambda (h)
    ((lambda (x) (h (delay (x x))))
     (lambda (x) (h (delay (x x)))))))

(define h
  (lambda (delayed-f)
    (lambda (n)
      (if (zero? n)
          1
          (* n ((force delayed-f)
                (sub1 n)))))))

(define fac (y h))
```

実際、このようにすると無限ループに陥ることなく階乗関数facを定義することができて、(fac 2), (fac 3), (fac 4),... は、2, 6, 24, ... と正しい答えを返す。

参考文献

[1] Clinger, W. et. al. : Revised^3 Report on the Algorithmic Language Scheme, AI Memo 848a (1985), MIT Artificial Intelligence Laboratory.

[2] 松田裕幸：SCHEME－抽象化能力をもつgoto-less手続き型言語－、情報処理、vol. 35, pp.227-233, 1994、情報処理学会

[3] Springer, G and D. P. Friedman: Scheme and the Art of Programming, Chap. 16-17, pp. 515-578, MIT Press(1989)

[4] Abelson, H. et. al. : Structure and Interpretation of Computer Programs, Chap. 5, pp. 383-503, MIT Press(1985)

[5] 後藤滋樹：記号処理プログラミング、第4章3(a)節、pp.181-187、岩波書店 (1988)

[6] 井田哲雄：計算モデルの基礎理論、第4章8節、pp.114-119、岩波書店 (1991)