Bulletin of Aichi Univ. of Education, 58（Natural Sciences）, pp. 21−25, March, 2009

Objective-C による Lisp 言語処理系の試験的実装

安本太一

情報教育講座

An Experimental Implementation of Lisp System by Objective-C

Taichi YASUMOTO

Department of Information Sciences, Aichi University of Education, Kariya 448 - 8542, Japan

1 はじめに

Objective-C を用いて試験的に Common Lisp[1] 言語処理系を実装しているので，報告する．

Lisp 言語処理系は，長い間，C 言語で記述するのが常道であった．C 言語で記述するようになる前は，実行効率やメモリ効率の理由で，アセンブリ言語で記述していた［2, 3, 4, 5］．C 言語で記述とうたっている言語処理系でも，当初は，実行効率やメモリ効率を重要視する部分は，その部分だけアセンブリ言語で記述するということが行われていた. そのような状況も, CPU の高速化やメモリの低価格化大容量化により，全て C 言語で記述しても実用上問題なくなった．

処理系の開発者の立場からは，低級な言語よりも，高級な言語の方が，開発が容易であることはいうまでもない．開発が容易であることは，処理系の信頼性向上にも結びつく．事実，Common Lisp 言語処理系である Kyoto Common Lisp（以下 KCL という）［2, 3］は，C 言語と Common Lisp 自身で開発されている．Common Lisp 言語で記述された部分は，最終的に KCL のコンパイラで C 言語に翻訳されるようになっている. Lisp 言語に比べて低級な C 言語だけでは, Common Lisp の組み込み関数を書ききれないというのが実情であろう．

今回，処理系記述のためのプログラミング言語として高級なものを追求していく姿勢の一つとして，オブジェクト指向とごみ集めの機能を備えたプログラミング言語である Objective-C 言語を採用することを試みた．具体的には，KCL の C 言語で記述されている部分を，Objective-C 言語で書き換えることを試みた．KCL においては Lisp 言語処理系として基礎的な部分が C 言語で記述されており，便宜上，単に KCL を Objective-C 言語で書き直すといっても差し支えない．

2 Objective-C 言語について

Objective-C 言語は，C 言語にオブジェクト指向の機能を追加したものである［6, 7, 8］. C 言語にオブジェクト指向の機能を付け加えたものには C＋＋言語もあるが，C＋＋言語（や Java 言語）に比べると，Objective-C 言語は非常に動的である．Objective-C 言語のコンパイラはオブジェクト指向の機能のことについて，多くのことを実行時まで決めずにおく特徴がある．例えば，メソッド探索は，コンパイル時ではなく，実行時に行われる．例えば，コンパイラは，ソースコード上でありそうもないメソッドを使用しようとしている場合，「このオブジェクトはこのメソッドに反応しないかもしれない」といったような警告をだすだけでエラーにはしない．このような Objective-C 言語の動的な性質は，動的な型付けを行う Lisp 言語の実装に利用できる可能性がある．

Objective-C 言語のために C 言語に追加された構文はわずかで簡潔であり，Objective-C 言語は C＋＋言語よりはるかに修得しやすい［9］. Lisp 言語処理系を，Objective-C 言語で実装したとしても，その処理系の保守が困難になることはない．

Objective-C 言語は，従前から，リファレンスカウント方式のメモリ管理を搭載していた．インスタンスオブジェクトが何か所から参照されているかをカウントしておき，どこからも参照されなくなったら（カウンタがゼロになったら），そのインスタンスオブジェクトが使用していたメモリ領域を解放するというものである. リファレンスカウント方式は, 相互参照になってしまったオブジェクトのメモリ領域を回収できない，カウントの仕方を間違えるとメモリークやぶら下がりポインタを引き起こすといった問題点がある．

Objective-C2. 0言語からは，マークアンドスィープ方式のごみ集めの機能が実装された．Objective-C2. 0 言語のマークアンドスィープ方式のごみ集めは，ルート集合（外部変数，自動変数，スタック）からたどれないインスタンスオブジェクトが使用していたメモリ領域を解放する．厳密には，ルート集合に含まれるのは，NSObject およびそのサブクラスのインスタンスへの参照だけであり，ポインタを含む C 言語オリジ

ナルのデータは，外部変数，自動変数，スタックにあってもルート集合に含まれない．

Objective-C 言語は，Apple 社の UNIX ベースの OS である MacOSX の公式開発言語であり，MacOSX 上のアプリケーションフレームワーク Cocoa の中核言語として採用されている．今後，特に断りがなければ，Objective-C 言語とは Objective-C2.0言語のことを指すことにする．簡潔のために，Objective-C 言語を単に Objective-C と記述する．

3 Objective-C による Lisp 言語処理系実装の意義

C 言語で Lisp 言語処理系を記述することがある程度成功している今日において，Objective-C で Lisp 言語処理系を記述する理由は以下のとおりである．

1．美しくない実装の解消

C 言語で実装された多くの Lisp 言語処理系においては，Lisp の各データ型は構造体で定義され，これらの Lisp のデータ型を総称するものとして共用体が用いられ，Lisp データ（セル）へのポインタはこの共用体へのポインタで表現される［2, 3, 5］．これらのデータ型の構造体の先頭にはデータ型を示すタグ領域がある．このタグをみることによって，共用体へのポインタとして渡された Lisp データのデータ型を判定する．Lisp データを受け取った時は，このデータ型のデータを判定するために，タグ領域だけを持ったダミーの構造体やコンスセルへの構造体をとりあえず使ったりなど，実害はないが，必ずしも美しいとはいえない実装が存在する．このような必ずしも望ましくない実装を，オブジェクト指向の機能を備えたプログラミング言語を使うことによって解消できる．

2．OS が備えているライブラリの利用し易さの向上

Lisp 言語処理系では，処理系そのものの動作や，Lisp 言語が得意とするリスト構造の操作において，使用されなくなるデータ（ごみ）が生じる．これらのデータが使用していたメモリ領域を再利用するために，ごみ集めが必要となる．これまでの Lisp 言語処理系では，たいてい，処理系自身がごみ集めの機能を有していた．処理系自身が備えているごみ集めの機能は，メモリ管理が処理系内部で閉じているのが普通である．Lisp 言語の仕様の範囲のみでプログラムを作成する場合はこれで困らないが，外部のライブラリ関数を使う場合は，これらのライブラリ関数のメモリ管理との共存を考えなければならない．筆者の過去の経験によると，外部のライブラリ関数が OS に備わっている動的メモリ割当の malloc を使用しているならば，処理系のメモリ管理部に malloc シミュレータを用意する必要があった．実行ファイルを作成するリンク時には強制的に malloc シミュレータの方の malloc にリンクさせるのである．このような対処は，どのようなメモリ管理を行っているかを把握する必要があるので，外部のライブラリ関数のソースコードがないと一般には難しい．

Objective-C とそのランタイムにはごみ集めの機能が実装されている．MacOSX 用のアプリケーションを開発するためのアプリケーションフレームワーク Cocoa を使ったアプリケーションのメモリ管理も，このランタイムの方で行うことになっている．Lisp 言語処理系のごみ集めを Objective-C とそのランタイムに任せると，ごみ集めの主導権がこのランタイムの方に移るが，OS が用意しているフレームワークを Lisp 言語の方から使用しようとするときに，メモリ管理についてほんとんど心配しなくてよくなる．これは，GUI などを使った実用的なアプリケーションの開発を Lisp 言語で行うことを容易にするし，Objective-C で記述されたプログラムと Lisp 言語で記述したプログラムを結合することを容易にする．

3．処理系の信頼性向上

ごみ集め等の Objective-C の機能を使うことよって，C 言語やアセンブリ言語で記述された場合に比べて，Lisp 言語処理系の信頼性が向上する．例えば，Objective-C のごみ集めの機能を使えば，組み込み関数や特殊形式のコーディングに際して，ごみ集めにより回収されると困る一時的なデータを（ごみ集めのマークの起点となっている）スタックに退避するといった配慮をすることを忘れる危険性が全くなくなる．KCL を例にとれば，このような一時的なデータの退避は随所にみられるし，KCL のコード修正履歴にもこのよう退避忘れの修正が多くあり，ごみ集めの利用一つをとっても，処理系の信頼性向上につながる．

その他の例を挙げると，処理系の国際化（多言語化）は，処理系独自で行うより，国際化に対応している Objective-C のクラスを使う方が得策である．国際化には多くのノウハウが必要であり，労力をかけて独自に開発しても，バグが消えて安定するまでに時間を要するであろう．例えば，Objective-Cの文字列クラスNSStringやNSMutableStringは国際化に対応しており，日本語に関しては EUC，Shift-JIS などへの変換もサポートしている．

ここに挙げたように従前は処理系の開発者が独自に対応しなければなかなかった機能も，開発環境や実行環境（例えば Objective-C と MacOSX の

アプリケーションフレームワーク Cocoa）によってあらかじめ実現されているものがある．開発環境や実行環境の方で用意されているものを利用した方が，処理系のコード量が減ることもあり，処理系の信頼性は向上する．

4．教材や資料としての価値

Lisp 言語処理系の実装には，プログラミング言語の様々な機能や多くの API（Application Programming Interface：システムコール，ライブラリやアプリケーションフレームワークを使うための窓口）を使う．C 言語で実現された KCL のソースコードには，ポインタ，マクロ，動的なメモリ割当，ファイル入出力，例外処理などがあらわれていた．したがって，Lisp 言語処理系のソースコードは，プログラミング言語処理系の教材としてはもちろんのこと，プログラミングの教材ともなる．実際，いくつかの大学では，KCL のソースコードの読み会が行われており，KCL のソースコードをそのものが学習用の教材として活用されている．また，C 言語ソースコード付きの Lisp 言語処理系実現の本も出版されている [5]．

上記のことから，Objective-C で記述された Lisp 言語処理系のソースコードも，計算機科学を専門とする人たちの教材となるであろう．本稿執筆時点で，日本において流通している，Objective-C に関する日本語書籍は数冊だけである [9, 10, 11]．これまでにソースコードが公開されている Lisp 言語処理系は，アセンブリ言語，C 言語，Java 言語で記述されたものが大半であり，Objective-C で記述された Lisp 言語処理系のソースコートは資料的な価値が十分あると期待される．

4 基 本 方 針

Objective-C によって Lisp 言語処理系を実現するにあたっての基本方針を次のように定めた．

1．Lisp の各データ型の表現

Lisp の各データ型を，それぞれ Objective-C のクラスとして定義する．実際の Lisp のデータは，このクラスのインタスタンスとする．このようにすることによって，等号の一つである関数 eq は，インスタンスオブジェクトのアドレスの比較で行うことができる．

Lisp の各データ型を総称する，つまり Lisp の各データ型の親クラスとなる LObject というクラスを定義する．LObject は NSObject のサブクラスとする．以上のことについて例をあげると，Lisp の代表的なデータであるコンスセルの定義のインタフェース部は，図 1 のようになる．

```
@interface LObject : NSObject
@end

@interface LCons : LObject
{
  LObject *c_cdr;
  LObject *c_car;
}
- (id)init;
- (LObject *)car;
- (LObject *)setCar: (LObject *) obj;
- (LObject *)cdr;
- (LObject *)setCdr: (LObject *) obj;
- (void)writeObject;
@end
```

図 1　コンスセルの定義

2．ごみ集めへの対処

先述のように，Lisp の各データ型は間接的に NSObject のサブクラスとなっているので，LObject（とそのサブクラス）へのポインタ変数から指されている限りは，ごみ集めによって不用意に回収されることはない．

逆にいえば，不要な Lisp データは保持しないよう，すみやかにポインタ変数からは参照しないようにする．つまりポインタ変数をすみやかに nil にする．

バイト列からなるデータは，malloc を用いるのではなく，（NSObject のサブクラスである）NSData あるいは NSMutableData のインスタンスとして確保する．

3．既存の処理系である KCL を C 言語から Objective-C で書き換え

C 言語と Lisp 言語で記述された KCL を，Objective-C と Lisp 言語で書き直す．全く新しい Lisp 言語処理系を作成する選択肢もあるが，教育や研究の対象として世間に広く知られている KCL を書き直す方が，ソースコードの資料としての価値がある．オリジナルの KCL を知っている者の利便性を考え，書き直す際は，ソースコードのソースツリー，データ型の名前，関数名，変数名などは，オリジナルの KCL のものを可能な限り踏襲する．

また，Common Lisp の仕様は膨大なので，多くの人に利用されバク等も枯れている KCL の基本設計を利用することは重要である．基本設計はそ

のままで，オブジェクト指向のプログラミング言語を使ってLisp言語処理系を書き換えた場合，処理系の性能がどのように変化するのかは，貴重なデータとなる．

4．オブジェクト指向の流儀に沿ったコードの記述

KCLのソースコードの雰囲気を残しつつも，オブジェクト指向に沿った実装を行う．例えば，先述のコンスセルの定義のように，コンスセルの構造の定義とメソッドの定義は，まとめて書くようにする．オリジナルのKCLでは，コンスセルの定義はファイルobject.hに，関数carやcdrの定義はファイルlist.dに，プリンタの定義はファイルprind.dに分散していた．オブジェクト指向の流儀に従ってLisp言語処理系をかくと，このようになるという例を示したい．

メソッド呼び出し（メッセージ送信）を使った方が，オブジェクト指向の流儀にかなっている場合は，関数呼び出しではく，メソッド呼び出しを積極的に使う．メソッド呼び出しのコストは，関数呼び出しのコストの1.8倍程度であるという報告[9]があるが，オブジェクト指向の流儀を優先した場合はどの程度の実行効率低下になるかは，貴重なデータとなる．

5．アプリケーションフレームワークの積極的な利用

日本語文字など多言語に対応した文字列クラス（NSStringやNSMutableString），辞書クラス（NSDictionaryやNSMutableDictionary）といった，アプリケーションフレームワークCocoaに備わっている有用なクラスを積極的に利用する．

過去の研究では，KCLに日本語文字処理機能を実現するために，文字に関するデータ型（文字型や文字列型など）とこれらのデータ型に関連する組み込み関数を拡張した[12]．多言語に対応するためのクラスやライブラリがOSに備わるようになった現在，処理系単独で多言語に対応する状況ではなくなった．

オリジナルのKCLではハッシュとリストをつかって実現していた記号表（名前から記号への対応表）も，辞書クラスを使えば事足りる．

KCLが実装された当時はOSが提供するライブラリやクラスが充実していなくてKCL自身が抱えていた部分を，KCLのソースプログラムから削除し，OSのライブラリやクラスにゆだねて，KCLの実行ファイルのサイズを小さくする．

6．組み込み関数のプラグイン化

Common Lispには，膨大な数の組み込み関数がある．Lispプログラムによってどのような関数を使うのか傾向も異なるであろうが，普遍的に使用頻度が低い関数があるのも事実である．iPhoneやiPod touchのような携帯用端末でLisp言語処理系を動作させるためには，必要メモリ容量が少ないことに越したことはない．Objective-Cには，当面必要なないクラスの集まり（モジュール）を実行ファイルとは別のファイルに格納しておいて，必要になったときに動的にロードする機能がある．このモジュールの動的ロード機能を用い，KCLの最小必要メモリ容量を少なくすることを試みる．

個々の組み込み関数を，コンパイルされた関数を総称するLCfunというクラスのサブクラスとしておけば，関数を動的にロードすることが可能である．ユーザが定義した関数をコンパイルして，ロードする時も，この動的ロード機能を使う．

5 現在の状況

Objective-CによるKCLの書き換えは，現在，一部の組み込み関数が動作するところまで進んでいる．組み込み関数＋のソースコードを，図2に示す．組み込み関数＋の実体は，LPlusクラスのインスタンスである．LPlusは，コンパイルされた関数を総称するLCfunクラスのサブクラスである．

LPlusのメソッドcf_selfが，オリジナルのKCLのLplus（＋の実体）に対応していて，その定義はほとんど同じである．一方，LPlus（Lplus）で使用している補助関数number_plusの定義は，大幅に変わった．オリジナルのKCLでは247行あったものが，今回Objective-Cを使って書き換えたものは4行である．オリジナルの場合は，引数x，yのデータ型の組み合わせ（例えば，双方とも整数，一方が整数で他方が浮動小数点数など）の場合分けによる処理がnumber_plus内で行われていたので，行数が多くなっていた．一方，Objective-Cを使って書き換えた場合は，［x add: y］というように，xにyを加えるというメッセージを送るだけになっているので，簡潔になっている．yのデータ型に対する対処はxのクラスに任せるというオブジェクト指向的な対応をしているからである．number_plusの方は，数値を表すデータ型としてどのようなものが実装されているかは関知しないように変わっている．例えば，まだ，数値を表すクラスのデータ型のうち，未実装のものがまだあっても，number_plusやLPlusのコンパイルはできるし，実行もできる．事実，本稿を執筆時点では，分数を表すデータ型のクラスの実装はできていない．

このソースコードには載せていないが，LPlusの記号＋への束縛は次のように行う．処理系起動時の初期化のところで，make_function（“＋”，“LPlus”）；というObjective-Cの関数を実行して，LPlusという名前を持ったクラスを探し，そのクラスのインスタンスを生

成し，そのインスタンスを記号＋に束縛するということを行っている．今回の実装は“LPlus”という部分は文字列となっていて，オリジナルの KCL では関数のアドレス定数 Lplus となっていたことと比べると，動的な実装となっている．

```
#import "object.h"

LObject* number_plus(LObject *x, LObject *y)
{
  return([x add: y]);
}

@interface LPlus : LCfun
@end

@implementation  LPlus
- (void)cf_self{
  int i,j;

  j = vs_top - vs_base;
  if(j == 0){
    vs_push(small_fixnum(0));
    return;
  }

  for(i = 0; i < j; i++)
    check_type_number(vs_base[i]);
  for(i = 1; i < j; i++)
    vs_base[0] = number_plus(vs_base[0],
                             vs_base[i]);
  vs_top = vs_base+1;
}
@end
```

図２　組み込み関数＋の定義

６　ま　と　め

現在行っている Objective-C による KCL の書き換えの様子を報告し，オブジェクト指向のプログラミング言語を使って Lisp 言語処理系を記述する実例を示した．Common Lisp の仕様は膨大なので，書き換え完了までに時間を要することが予想されるが，未完成な部分が多い中で，Objective-C の動的な性質により，とりあえず動ことができる例を示した．

書き換えが完了したら，メモリ性能や実行性能の評価実験を行い，その評価結果を報告する予定である．その際に，本稿では述べていないが，性能に影響を与えたと予測される部分の実装の詳細について報告する．筆者のところには，MacOSX 上で動作するオリジナルの KCL の，PowerPC 版と Intel 版，またこれらのそれぞれに対応して32ビット版と64ビット版があり［13, 14, 15］，実行環境として想定される全ての版について性能評価実験ができるので，C 言語で実装した場合と Objective-C で実装した場合の性能比較についてある程度普遍的な結論がだせる見込みである．

Lisp 言語から，GUI など担っているアプリケーションフレームワーク Cocoa のクラスのインスタンスにアクセスすることや，Objective-C のプログラムから Lisp のデータを扱う機能も付加することも今後の課題である．一つのプログラムを，Objective-C と Lisp 言語のそれぞれが得意とする場面で使い分けて，記述できるようにしたい．

参　考　文　献

［１］Steele, G. L. Jr.: *Common Lisp the language,* Digital Press (1984).

［２］Yuasa, T. and Hagiya, M.: *Kyoto Common Lisp Report,* Teikoku Insatsu Publishing (1985).

［３］Yuasa, T.: Design and Implementation of Kyoto Common Lisp, *Journal of Information Processing,* Vol.13, No.3, pp. 284-295 (1990).

［４］田中哲朗：SPARCの特徴を生かしたUtilisp/Cの実現法, 情報処理学会論文誌, Vol. 32, No. 5, pp. 684−690(1991).

［５］小西弘一，清水剛：C プログラムブックⅢ―Lisp 処理系の作成―，アスキー（1986).

［６］*Object-Oriented Programming with Objective-C,* Apple (2007).

［７］*The Objective-C 2. 0 Programming Language,* Apple (2008).

［８］B. J. Cox, and A. J. Novobilski: *Object-Oriented Programming Approach 2nd Edition,* Addison-Wesley (1991).

［９］萩原剛志：詳解 Objective-C2. 0，ソフトバンククリエイティブ（2008).

［10］B.J.コックス，A.J.ノボビルスキ共著，松本正雄訳：オブジェクト指向のプログラミング改訂第２版ソフトウェア再利用の方法，新紀元社（2004).

［11］木下誠：たのしい Cocoa プログラミング［Leopard 対応版]，ビー・エヌ・エヌ新社（2008).

［12］安本太一，湯淺太一：Kyoto Common Lisp の日本語文字処理機能の実現とその評価，コンピュータソフトウェア，Vol. 9，No. 5，pp. 12−24（1992).

［13］安本太一：Common Lisp 言語処理系の64ビット化，愛知教育大学研究報告　自然科学編，第五十三輯，pp. 22−32（2004).

［14］安本太一：Common Lisp 言語処理系による64ビット環境の評価，愛知教育大学研究報告　自然科学編，第五十五輯，pp. 9−13（2006).

［15］安本太一：Common Lisp 言語処理系におけるインテル Xeon5100の評価，愛知教育大学研究報告　自然科学編，五十七輯，pp. 25−29（2008).

（2008年９月17日受理）