記号処理 42－3
（1987. 9. 25）

konoCL － kono Common Lisp －

中村 輝雄　山本 昌彦　藤岡 秀樹
川上 敦　納富 雅人　吉田 晶子

日立ソフトウェアエンジニアリング株式会社 研究部

konoCLは、UNIX SYSTEM V*(MC68000)上で開発しているフルセットCommonLisp処理系である。トップレベルの評価関数や、スペシャルフォームの評価関数など、処理系のほとんどをCommonLisp自身で記述している。ユーザは、トップレベルを再定義したり、不要な関数を削除することにより、最適な環境でアプリケーションプログラムを、UNIXのコマンドライクに稼働させることができる。また、プログラム実行時に更新されないLispデータが、ゴミ集めに負担をかけないようなメモリ管理を行っており、大規模な辞書データベース等を扱うアプリケーションにも適した処理系である。

konoCL － kono Common Lisp －

Teruo Nakamura　Masahiko Yamamoto　Hideki Fujioka
Atsushi Kawakami　Masato Noutomi　Akiko Yoshida

Research & Development Department, Hitachi Software Engineering Co., Ltd.
6-81, Onoe-machi, Naka-ku, Yokohama 231, Japan

konoCL is a full set Common Lisp system being implemented on UNIX SYSTEM V* with MC68000 CPU. Most of the system, including top level evaluator and some of special form evaluators, are written in Common Lisp itself. KonoCL users can redefine the top level, can remove unnecessary functions,and can execute their application programs directly as a UNIX command in a suitable environment. Memory management is implemented so that immutable lisp objects do not affect garbage collection time, making konoCL suitable for applications with big but static dictionaries.

1.はじめに

筆者らは1983年から独自にLisp処理系[1](konoLisp)を開発し、これを使って今までに自然言語インターフェイスシステム[2][3]等の研究、開発を行ってきた。

一方、米国においては、Lisp言語の仕様の統一を目指して1984年にCommonLisp[4]が発表され、ここ2~3年の世界の動きからこのCommonLispが標準仕様になりつつある。従来からあるシステムもCommonLispで書き直され始めている。このため、筆者らで開発してきたシステムもさまざまなマシンで稼動させる必要性から、CommonLispに書き直し始めている。こうした背景のもと、1985年より筆者ら独自で CommonLisp(konoCommon Lisp、以後konoCLと呼ぶ)の開発を行ってきた。

ところで、Lisp言語でアプリケーションプログラム（以後APと呼ぶ）を開発した場合、従来のLisp処理系では次のような問題があった。

「APに不必要な組み込み関数がメモリ中に取り込まれて、必要以上にメモリを消費する。」

そこで、今回konoCLを開発するにあたりこの問題の解決を第一に考え、次のような設計方針を立て開発を行った。

(1)APを必要最小限で実行できる環境を提供する。

(2)UNIXマシンをターゲットとして、従来からのUNIX文化に溶け込ませる。

2. konoCLの構成

2.1 konoCLの基本構成

konoCLは MC68000系のUNIXマシン上で稼動する。konoCLは核になる KSPと、この上で稼動するCLiP、CLiCの総称である。この基本構成を図1に示す。

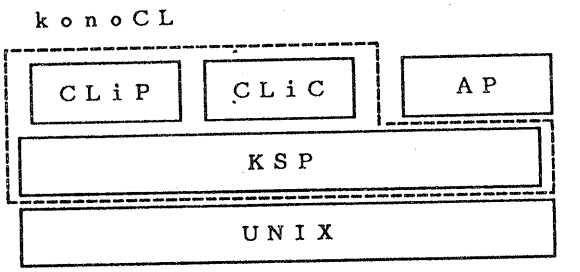

図1　konoCLの基本構成

2.2 KSP

(Kernel for Symbolic Processing)

KSPはkonoCLの核として、またScheme[5]等の他の記号処理言語の核としての意味を込めたものである。この KSPを、アセンブリ言語とC言語を使って開発している。

KSPの役割は次の三点である。

(1)メモリ管理

(2)ローダ

(3)システムサービス

2.2.1 メモリ管理

メモリ管理はリストや配列等基本的データの領域の確保及び解放を行う。特に、領域の解放はガーベジコレクション[6]で一括して行う。

2.2.2 ローダ

ローダはプログラムを実行可能な形式でメモリ中に取り込む。プログラムはつぎの三種類のファイル形式で格納されている。

(1)テキスト型ファイル (.lsp)

ユーザが作成するソースファイル

(2)ファスル型ファイル (.fsl)

CLiCが作成するオブジェクトファイル

(3)イメージ型ファイル (.img)

スナップショット(3.2.2で説明する)で作成されるメモリイメージファイル

2.2.3 システムサービス

システムサービスは基本的データの参照、更新、及び演算機能を提供する。例えば、リストのシステムサービスとしてcar、cdr及びrplaca、rplacdを提供する。konoCLでのAP開発はすべてこのシステムサービスを用いてLisp言語で行う。

2.3 CLiP (Common Lisp Interpreter)

これはCommonLispの関数群の総称である。CLiPの開発は、その下位の KSPの提供するシステムサービスを使ってLisp言語で行う。CLiPは KSPの最初のAPである。

2.4 CLiC (Common Lisp Compiler)

CLiCの開発も、CLiPと同様にその下位のKSPのシステムサービス用いる。CLiCはKSPの二番目のAPである。

2.5 AP

ユーザはkonoCLを使って、Lisp言語でAPを開発する。APはCLiPの上だけでなく、APで用いるLisp関数だけを取り込んで KSP上で直接動かすことができる。

図２は、CLiPの上でAPを動かしているときの状態であり、APには十分すぎるLisp関数がメモリに取り込まれており、メモリを浪費している。一方、図３は KSPの上で直接動いているときの状態であり、必要なLisp関数だけがメモリに取り込まれている。このように、 KSP上でAPを直接実行することで、メモリ効率を高めることができる。

ただし、必要最小限のLisp関数を取りだすリンカの実現方式を検討する必要がある。

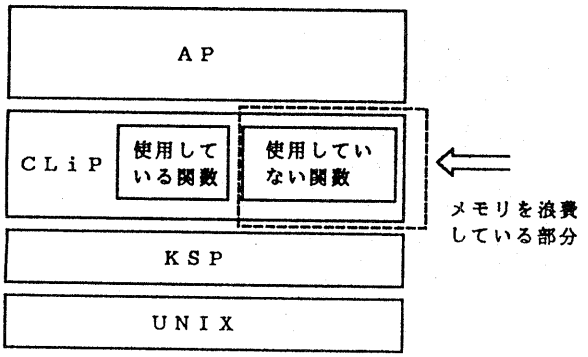

図２　ＣＬｉＰ上のアプリケーションプログラム

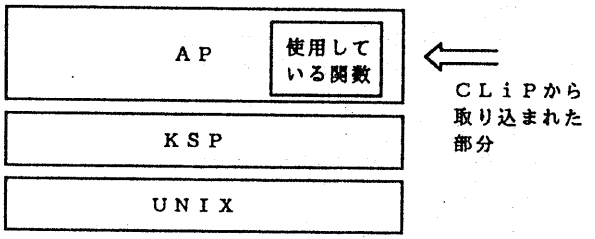

図３　ＫＳＰ上のアプリケーションプログラム

3. 小さな核 KSP

3.1 KSPの構成

KSPがメモリ管理、ローダ及びシステムサービスからなることは既に２章で述べた。これをさらに詳細に分けると図４のようになる。本章では、 KSPを小さくするための工夫について述べる。

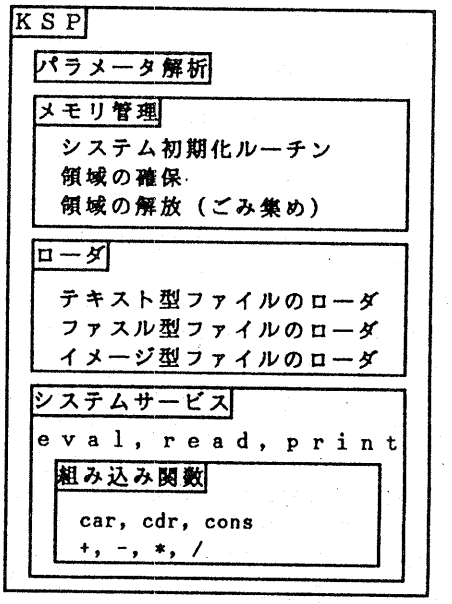

図４　ＫＳＰの構成

3.2 メモリ管理での工夫

メモリ管理の内、領域の確保及び解放を行うルーチンは、基本的であり KSPから取り除くことはできない。しかし、システム初期化ルーチンは削除ができる。これを説明するために、まずシステム初期化ルーチンの役割を述べる。

3.2.1 システム初期化ルーチンの役割

システム初期化ルーチンは、まず適当な大きさでメモリを図５のように分割し、その後システムが使うLispデータをシンボル領域とコンス・ベクタ領域に生成する。このシステムが使うLispデータにはNIL，T等のシンボルやリードテーブル、I/O 用ストリームがあり、その数は1000を越えている。このため、このルーチンはオブジェクトで約56KB(KSP全体の約10%)を占めているが、KSP起動直後の初期化が終わると以後使われることがなく、メモリの無駄になる。

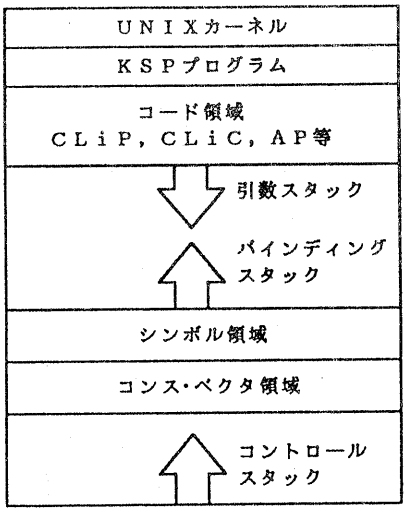

図５　ＫＳＰのメモリレイアウト

3.2.2 システム初期化ルーチンの削除

KSPではシステム初期化ルーチンを削除するため、スナップショットの機能を利用する。3.2.1 で述べたようにメモリを初期化したところで、メモリ状態をそのままファイルにセーブ(スナップショットと呼ぶ)し、次に KSPを起動する時はシステム初期化ルーチンを使わずスナップショットで生成したファイルをメモリに読みこんで初期化を行う。これにより、システム初期化ルーチンが削除できる。ただし、現在は KSPを試作している段階であり、システム初期化ルーチンの削除は行っていない。

3.3 システムサービスでの工夫

3.3.1 eval

インタプリタの核となるevalは、次の順で S式を評価(実行)していく。

eval[form, env]の実行

(1)formが定数の時
　form自身を値として返す。
(2)formがシンボルの時
　シンボルの値を返す。
(3)(1)、(2)以外の時、formは
　(fun arg1 arg2 - - - argn)の形である。
　(a)funが関数の時
　　引数argを順番に評価して、関数funを呼び出す。
　(b)funがマクロの時
　　formとenvを引数としてfunのマクロ展開関数を呼び出し、返ってきた値をformにセットして、再びevalを実行する。
　(c)funがスペシャルフォームの時
　　24個のスペシャルフォーム(quote、setq 等)に対応して、24種類の評価の方法を行う。

ここで、evalをすべて KSPに組み込むと上記の分類に対応したルーチンが必要となる。特に、スペシャルフォームでは24個の異なる評価関数が必要となる。しかし、一般にAPは、コンパイルした機械語プログラムで実行するため、Lispインタプリタの核であるevalは不要となる。そこで、 KSPのevalはＳ式に対応した評価関数を呼び出す機能のみを持たせ、実際の評価は主にLisp言語で実現した評価関数で行うことにした。新しいevalは次のようになる。

eval[form, env]の実行

(1)formが定数の時
　form自身を値として返す。
(2)formがシンボルの時
　シンボルの値を返す。
(3)(a)funがコンパイルした関数の時
　　(system::exec-compiled-code form env) を実行する。
　(b)funがコンパイルしていない関数の時
　　(system::exec-user-function form env) を実行する。
　(c)funがマクロの時
　　(system::exec-macro form env) を実行する。
　(d)funがスペシャルフォームの時
　　24個のスペシャルフォームに対応した、
　　(system::exec-quote form env)
　　(system::exec-let form env)
　　- - - - - - - - -
　　を実行する。

例えば、quoteの評価関数であるsystem::exec-quoteは次のように実現できる。ただし、system::exec-quoteは現在は KSPに組み込んでいる。

```
(defun system::exec-quote (form env)
  (car (cdr form)))
```

このように評価関数をLisp言語で実現し、KSPから取り除くことで、 KSPのevalがかなり小さくなる。表１に KSPに組み込んでいる評価関数を示す。将来は、すべての評価関数をLisp言語で実現する予定である。

表３．１　評価関数の開発言語

項番	スペシャルフォーム	言語	項番	スペシャルフォーム	言語
1	block	Lisp	13	macrolet	Lisp
2	catch	asm	14	multiple-value-call	C
3	compiler-let	C	15	multiple-value-prog1	Lisp
4	declare	C	16	progn	Lisp
5	eval-when	Lisp	17	progv	C
6	flet	Lisp	18	quote	C
7	function	C	19	return-from	Lisp
8	go	Lisp	20	setq	C
9	if	C	21	tagbody	Lisp
10	labels	Lisp	22	the	Lisp
11	let	C	23	throw	asm
12	let*	C	24	unwind-protect	asm

表３　KSPの出力関数のサポートしているデータ型

項番	データ型	KSPの出力関数のサポート状況
1	integer	fixnumのみフルサポート
2	ratio	サポートせず
3	floating-point number	フルサポート
4	complex	サポートせず
5	character	フルサポート
6	symbol	フルサポート
7	string	simple stringのみフルサポート
8	cons	サポートせず
9	structure	サポートせず
10	array	サポートせず
11	stream	サポートせず
12	interpreted closure	サポートせず
13	compiled closure	フルサポート
14	compiled code	フルサポート
15	external closure	フルサポート
16	binary	フルサポート
17	TSPECIAL	フルサポート
18	TUNBOUND	フルサポート

3.3.2 read

CommonLispのreadは、言語仕様上一文字づつ処理する[2]。そこで、 KSPではreadのドライバのみを提供し、リードマクロをサポートする関数は KSPからはずした。(表2参照)

表2 リードマクロの開発言語

項番	開発言語	リードマクロの数
1	Ｌｉｓｐ	２１
2	C	１１

3.3.3 print

CommonLispの printは非常に強力な機能を提供しているが[4]、すべてのAPで利用されるとは考えられない。そこで、基本的な部分のみを KSPで提供し、それ以外の機能は KSPからはずした。例えば、コンスや配列等の再帰的なデータ型の表示は KSPでは行っていない。(表3参照)

3.3.4 組み込み関数

CommonLispには 約600個の組み込み関数がある[4]。しかし、これらの関数の内の幾つかはより基本的な関数のAPとみなすことができる（例えば、cadr、cddr等は、car、cdrで実現できる）。従って、 KSPでは核になる約100個の関数を提供することにし、それ以外はCLiP自身で記述した。

4. UNIXでのKSP

4.1 コマンド仕様

UNIX上で KSPを稼動させるためには、コマンド仕様もUNIX文化に溶け込んだものでなければならない[7]。例えば、オプションの指定は－記号を用いる必要があるし、ファイルのリダイレクションも行える必要がある。このようにして決めたコマンド仕様を付録に示す。ここでは、 KSPの工夫点を述べる。

4.2 APのロード

３章で述べたように、 KSPには必要最小限の機能しかない。このため、APを動かすには、 KSPにプログラムをまずロードしてもらう必要がある。-fオプションはこの機能を提供する。例えば二つのファイル、foo.fsl、bar.fslというからなるAPの起動は次のコマンドで行う。

```
$ ksp -ffoo.fsl,bar.fsl
```

また、CLiPも KSPのAPであり、この場合はつぎのコマンドで行う。

```
$ ksp -fclip.fsl
```

この機能を使えば、konoCLでProlog処理系を実現した場合は、次のコマンドで実行できCommonLispインタプリタはいらない。

```
$ ksp -fprolog.fsl
```

4.3 引数の受渡

APによっては引数を必要とするものもある。 KSPではＣ言語を参考にしてこの機能を実現した[7]。

ユーザが次のようにコマンドを入力した場合を考える。

```
$ ksp arg1 arg2 arg3
```

このとき、 KSPはシステムの初期化を終えた後、*main*という関数にarg1、arg2、arg3を文字列にして渡す。例えば、コマンドの引数をディスプレイに表示するAPは次のように実現できる。

foo.lspファイルの内容

```
(defun *main* (&rest l)
  (dolist (x l) (print x) ) )
```

```
$ ksp -ffoo.lsp arg1 arg2 arg3
```

ただし、正確には foo.lspをロードするだけでは、APを実行することができない。このことは、6章で述べる。

4.4 環境の設定

KSPはいろいろな環境（シンボル領域の大きさやスタックの大きさ等）を設定するパラメータがある。これらのパラメータをいちいちコマンドで与えるのではなく、UNIXのshやviコマンドと同様に.ksprcファイルを用いて各種パラメータの設定を行うことを検討している[7]。

4.5 その他の注意事項

UNIX上で KSPを開発するためのその他の注意事項として以下のことが挙げられる。

(1) KSPの終了はexit(2)を使う。
(2) UNIXの標準入出力ファイルを使う。
(3) プロンプトはユーザ定義とする。

5. 評価

5.1 メモリ効率

KSPを小さくしたことによるメモリ効率の向上を評価する。表４は KSPとCLiPの使用するメモリ量を示している。

表４　メモリ量の比較　（単位：バイト）

項番	システム名	最低限必要とするメモリ量
1	ＫＳＰ	５６０Ｋ
2	ＣＬｉＰ	１，４００Ｋ

この表から判断して、もしCommonLispのすべての関数を KSPで実現したとなると 1,400KBのメモリが必要となり、この上で動くAPは最低でも 1,400KB必要である。しかし、このように KSPを小さくしたことで

560ＫＢ
＋(APのサイズ)
＋(APで使用する関数群のサイズ)

ですむようになった。ただし、APで使用する関数群のサイズはAPの種類による。

5.2 CLiPの起動時間

KSPを小さくするため、 KSPが起動時に -fオプションで指定されたファイルをロードしてCLiPの環境を整えるようにした。このため、すべての関数を KSPで実現したときと比べてシステム全体の起動が遅くならないかという疑問が出る。表５はCLiPの起動時間を示している。

確かに -fオプションのようなファイルを一つ一つロードしてくる方法ではシステムの起動は非常な時間がかかる。しかし、スナップショットを利用した-rオプションを用いることで十分実用的な時間を得ることができた。

表５　ＣＬｉＰの起動時間（単位：秒）

項番	オプション	起動時間
1	-f	３９．０
2	-r(reloc)	２．２
3	-r	１．４

(注１)HP9000/350(CPU MC68020,25Mhz)で測定している。
(注２)-fは、１９１ファイルをロードしている。
(注３)-r(reloc)は、メモリマップを変更し、ロード時にリロケーションの必要な場合である。
(注４)-rは、ロード時にリロケーションの不必要な場合である。

6. 今後の課題

6.1 KSPの大きさ

５章で評価したように、 KSPの大きさはCommonLispの機能をすべて取り込んだCLiPに比べると非常に小さいが、Ｃ言語等の他のプログラミング言語からみればまだかなり大きい。これは、 KSPのシステムサービスの大きさによっている。これを解決するために、システムサービスも KSPからはずし、必要なときに必要となるシステムサービスだけを取り込む機構が必要である。

6.2 リンクの方法

引数をリストにしてプリントする次のプログラムを考える。

foo.lspファイルの内容

```
(defun *main* (&rest l) (print l))
```

```
$ ksp -ffoo.lsp arg1 arg2 arg3
```

このプログラムの実行は上のコマンドで行えるが、ここで問題が生じる。つまり、print 関数が KSPで提供されていないため、このファイルだけを-fオプションでロードしても実行できないのである。ここでリンカの必要性が出てくる。リンカは、print 関数に必要な関数だけをCLiPから取りだして foo.lspにリンクできなくてはならない。リンカの実現方式は、これからの検討課題である。

7. 参考文献

1) 角田 透 他:CPM-68K用Lisp(Kono Lisp)のLisp言語による開発:情報処理学会記号処理研究会資料31-12(1985)
2) 松島 利幸 他:ポータビリティを目指した日本語インターフェイスの試作: 情報処理学会自然言語処理研究会資料54-4(1986)
3) 松島 利幸: 多重領域をサポートした意味解析言語: 情報処理学会自然言語処理研究会資料61-3(1987)
4) Steele、G. L.: COMMON LISP: Digital Press (1984)
5) Ress、R. 他:Revised³ Report on the Algorithmic Language Scheme: The M.I.T. AI memo(1986)
6) Baker、H. G.: List-processing in real time on a serial computer:Commun. ACM 21-4(1978)

7) HP-UX Concepts and Tutorials Vol.5
Shell and Miscellaneous Tools: Hewlett
Packard(1985)

＊：ＵＮＩＸオペレーティングシステムは、米国ＡＴ＆Ｔ社が開発したソフトウェアであり、ＡＴ＆Ｔ社がライセンスしています。

付録

KSP(1) KSP(1)

NAME
ksp - kernel for symbolic processing

SYNOPSIS
ksp [options] arg1 arg2 ...

HP-UX COMPATIBILITY
Level: HP-UX/NON-STANDARD

Origin: SK

DESCRIPTION
Ksp is a kernel for symbolic processing. *Ksp* invokes *MAIN* function with arg1, arg2 ... If *MAIN* isn't defined, *ksp* quits.

The following options are recognized:

-bbsize Set the size of code area to *bsize*.

-ccsize Set the size of control stack to *csize*.

-d Debug mode for system implementors.

-efile Intern symbols in *file* as external symbols of LISP package.

-ffile1[,*file2*,...,*fileN*]
Load *file1*, *file2*, ... and *fileN*. There are two kinds of files.

(1) Files whose names end with **.fsl** are taken to be fasl files.

(2) All other files are taken to be text files. Their names probably end with **.lsp**.

-hhsize Set the size of cons vector area to *hsize*.

-kksize Set the size of argument stack to *ksize*.

-rfile Bigbang by *file*.

-sssize Set the size of symbol area to *ssize*.

-v Print the system parameters. If this option is used, *ksp* doesn't invoke *MAIN*.

SEE ALSO
clip(1) clic(1).