

LISP50システムとそのコンパイラ

太田 義勝 船垣 康善 (三重大・工) 吉田 雄二
(名大・大型計算機センタ) 福村 晃夫 (名大・工)

§1. まえがき

LISP50は、FACOM230-38上に、FORTRANで実現された、会話型LISPシステム(LISP38)を、OKITAC system 50/49上に、移植したシステムである。

LISP50は、FORTRANという、高級言語で書かれている為に、システムの、他計算機への移植性の面ですぐれ、また、システムの変更修正が、容易であるという特色を有するが、実行速度が、遅いという欠点を、持つ。

LISP38から、LISP50への移植にあたり、いままでのインタプリタのみのシステムに、コンパイラ・システムを加え、実行速度の改善を、はかった。

このコンパイラ・システムは、LISPで書かれたコンパイラと、インタプリタのもとで動作する仮想機械とから構成されている。コンパイラは、この仮想機械のコードを生成するように、作られている。

本報告では、LISP50のコンパイラ・システムについて、特に、仮想機械と、コンパイラの構成について、述べる。また、インタプリタとの実行速度の比較等の性能評価の結果について、報告する。

§2. LISP50システムの概略

LISP50は、Interlisp Eモデルにした会話型LISPシステムである。

LISP50は、OKITAC system 50/49 のQS上で、稼動している。LISP50の入出力機器構成を、図-1に、示す。LISP50のシステムの規模は、図-2のXメモリマップが、示す様に、フリー・スペースが、よかれ、スタック領域が、3kw、文字アトム領域が、4kw、文字アトムのプリント・ネームおよびコンパイラ・オブジェクトをロードする領域が、4kwで、ある。

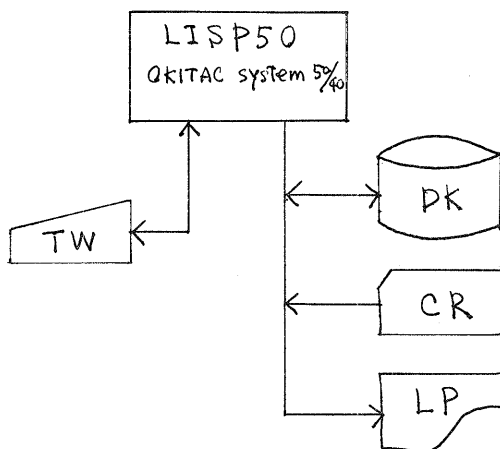


図-1 入出力機器構成

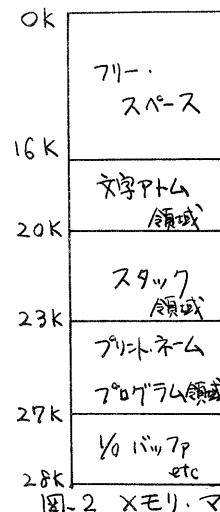


図-2 Xメモリマップ

LISP 5 のシステムの特色の一つとして、LISP プログラム開発の容易なシステムである、ということが、あげられる。そのためのリールとして、LISP で書かれた editor, pretty-printer, pre-processor, 関数値の呼び出し関係を図示するプログラム等が、用意されている。

付録 1 に、LISP 5 の組み込み関数の一覧表が、あげてある。詳しくは、文献(1) E. 参照のこと。

§ 3. コンパイラシステムの作成

コンパイラシステムの設計にあたって、目標としたのは、以下のことである。

- 1) 既存のインタプリタシステム上で、便えること。
- 2) 既存のシステムに、大きな変更を加えることなく、実現されること。
- 3) プログラムサイズが、大きくなりすぎないこと。(ユーザが、メモリ上で、使用出来るメモリは、48K に、限られる。データサイズを、大きくする為の要請)
- 4) インタプリタとの速度比が、大きいこと。

設計目標より、コンパイラシステムの構成を、次の様に、決定した。

- ① コンパイラは、LISP で、書く。(FORTRAN で、LISP コンパイラを、テスト的に、作成したが、メモリの問題で、採用しない。)
- ② コンパイラは、仮想機械のコードを、出力する。また、仮想機械は、FORTRAN で、記述し、インタプリタの実行時に、呼び出せる様にする。
- ③ 仮想機械は、スタック機械とする。ただし、効率の向上の為に、スタックのトップを、レジスタと見なして、スタック操作を、減少させる。
- ④ LISP で書かれたコンパイラと、FORTRAN で書かれた仮想機械の、インターフェースとして、システム内のプログラム領域に、コンパイラのオブジェクトをロードするロードを、LISP の関数として、組み込む。

§ 4. 仮想機械

コンパイラシステムも、FORTRAN で、全て記述するために、コンパイラの出力するオブジェクトは、QK1TAC system 50/40 の機械語ではなく、仮想的な、スタック機械の、仮想命令である。仮想機械を、FORTRAN で、記述することにより、ポータブルな、コンパイラシステムが、出来上がった。

以下に、仮想機械の構造、仮想命令について、述べる。

§ 4.1 仮想機械の構造

仮想機械は、1種のスタック機械である。そのスタックのトップの要素は、レジスタ(R) 上に、置かれる。スタックへの PUSH 操作は、レジスタが、使用されていない(以下、空であるという)ときは、レジスタへ、PUSH する要素を置き、使用中の時には、レジスタの内容を、スタック本体に、押し込んでから、レジスタに、PUSH する要素を置くことにより、行なわれる。また、POP 操作は、単に、レジスタを、空と、見なすことにより、行なわれる。

この様に、スタックを、構成することによって、スタックポインタの操作、配列の参照を、減少させることが、出来る。このことが、オブジェクトの実行速度の向上に、つながっている。(普通のスタック構成の、仮想機械と比較して、10%以上の効率の向上が、認められた。)

仮想機械は、いま述べたレジスタの他に、スタック本体のトップを、示すスタックポインタ(TOP)、現在実行中の関数の、ラムダ変数の、スタック上での位置を示す、ベースレジスタ(BASE)、現在実行中の仮想命令の、プログラム領域中での位置を示す、プログラムカウンタ(PC)、subrの、カ1引数を置くためのレジスタ(R2)を、持つ。

現在実行中の関数の、ラムダ変数は、 $BASE+1, BASE+2, \dots, BASE+n$ のスタック位置にある。(nは、ラムダ変数の個数) $BASE+n+1$ と、 $BASE+n+2$ のスタック位置には、それぞれ、いま実行中の関数を呼んだ関数のBASE、その関数への戻り番地が、置かれている。

仮想機械は、インタプリタと、フリースペース、スタック、レジスタ(R)文字アトムセル(コンパイルされた関数の、タイプ、オブジェクトの先頭番地、引数の個数などが、格納されている。)を、インタプリタと、共有している。

これらの関係を、図3に、示す。

仮想機械は、インタプリタ実行時に、関数タイプ $lexpr$ (コンパイルされてロードされた関数を示す。)の関数を、applyする時に、インタプリタから、制御が、移され、実行される。仮想機械の実行が、終了すると、制御は、インタプリタに、戻される。

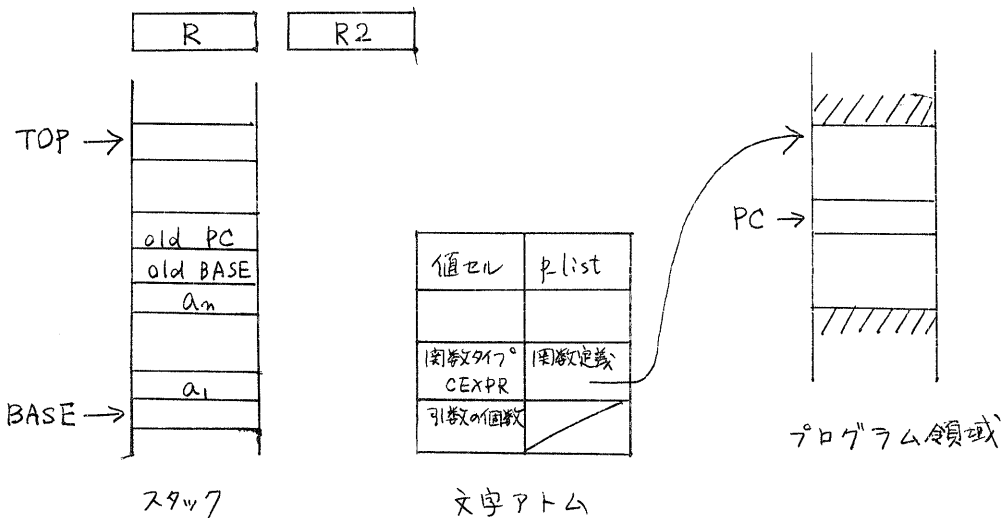


図-3 仮想機械のデータ構造

§ 4.2 仮想命令

仮想機械の実行する仮想命令は、全部で、21種類ある。表-1に、仮想命令の名称、記号、引数のタイプ、意味を、示す。

表-1 仮想命令

命令	記号	引数	意味
load immediate	LI, PLI [*]	S-expr	$R \leftarrow S\text{-expr}$
load local	L, PL [*]	d	$R \leftarrow \text{stack}(\text{BASE} + d)$
load global	LG, PLG [*]	v	$R \leftarrow \text{value}(v)$
jump	J	l	$PC \leftarrow l$
Nil jump	NJ	l	if $R = \text{Nil}$ then $PC \leftarrow l$
T jump	TJ	l	if $R \neq \text{Nil}$ then $PC \leftarrow l$
store local	ST	d	$\text{stack}(\text{BASE} + d) \leftarrow R$
store global	STG	v	$\text{value}(v) \leftarrow R$
mark	MK, PMK [*]		$R \leftarrow \phi$ "mark"
bind	BIND	n	for $i := 1$ to n do $\text{PUSH}(\text{Nil})$
call expr	C, PC [*]	fn	$\text{PUSH}(\text{BASE});$ $\text{PUSH}(PC);$ $\text{BASE} \leftarrow \text{TOP} - \text{nargs}(\text{fn}) - 2;$ $PC \leftarrow \text{def}(\text{fn})$
call subr	C ϕ , C1	f	ϕ , 1 引数 subr の呼び出し
	C2	f	$R2 \leftarrow \text{POP}();$ 2 引数 subr の呼び出し
call subr*	CN	f	$\text{PUSH}(R);$ subr* の呼び出し
return	RT	n	$\text{temp} \leftarrow \text{BASE};$ $\text{TOP} \leftarrow \text{BASE} + n + 2;$ $PC \leftarrow \text{POP}();$ $\text{BASE} \leftarrow \text{POP}();$ $\text{TOP} \leftarrow \text{temp}$

(注)

*印のついた命令は、命令実行の直前に、 $\text{PUSH}(R)$ を、実行する。

d: BASE からの offset

l: 番地

v: 変数名

fn: 関数名

f: 関数の通し番号

PUSH, POP: スタックへのプッシュ, ポップ

stack: スタック

value: 文字アトムa値セル

nargs, def: 関数の引数の個数と

コンパイルされたコードの先頭番地

§ 5. コンパイラ

LISP 5.0 のコンパイラは、LISP 関数 `compile[x]` として、実現されている。`compile[x]` は、引数として、コンパイルする関数名のリストをとり、値として、(関数名 ラムダ変数リスト オブジェクトリスト) の形をした、リストを、要素としてもつ、リストを返す。ここに、オブジェクトリストは、仮想命令を `op`、その引数を `arg` としたとき、`(op . arg)` を、要素とするようなリストである。

`compile[x]` が、ラムダ式で、定義された、LISP 関数を、仮想命令に、変換する時の規則を、表-2 に、示す。

仮想機械が、使用するスタックのトップが レジスタとなっているために、普通のスタック機械に対するコンパイラと比較して、ロード命令、コール命令の生成が、少し複雑になっている。

① ロード命令

変数の値などを、レジスタに、持ってくるロード命令は、レジスタが、その時点において、使用中であれば、`PLI(PL, PLG, PMK)` となり、レジスタが、空であれば、`LI(L, LG, MK)` となる。(レジスタは、`NJ`, `TJ` 命令の実行後、又は、`PROG` のステートメントを、評価し終った時に、空となる。)

レジスタが、使用中か、空か、は、コンパイラが、コンパイルを、行ないながら、レジスタのシュミレートを行なうことにより、決定される。

② コール命令

`cexpr`, `subr*` は、引数が、すべてスタックの中にある状態で、また、`subr` は、引数が、レジスタ (`R` と `R2`) に、のっている状態で、呼ばれる。関数の値は、レジスタ (`R`) に、返されることも併せて、コール命令は、関数タイプおよびレジスタの状態で、分かれる。

`cexpr` では、0 引数で、レジスタが、空の時以外は、レジスタの内容を `PUSH` してから、呼ばれる。(PC 命令) `subr*` は、いつでも、レジスタの内容を `PUSH` してから、呼ばれる。(CN 命令、少なくとも、"mark" が、レジスタ上にあるから) `subr` の場合は、中引数で、レジスタ使用中の時は、引数以外の値が、レジスタ上にあるので、関数値で、こめられない為に、`PUSH` してから呼ぶ。(Cφ 命令) 中引数、レジスタ空の時、または、1 引数の時は、何もせずに呼ぶ。2 引数の時は、レジスタ (`R`) に、第 2 引数、スタック上に、第 1 引数があるので、レジスタ (`R2`) に、ポップしてから呼ぶ。

§ 5.1 ローダ

コンパイラによって、作り出されたオブジェクトは、LISP 関数 `loader[x]` によって、プログラム領域に、格納される。この時、関数名に対応する、文字アトミセルの、関数タイプが `cexpr` に、関数定義が、ロードされたオブジェクトの先頭番地に、引数個数が、ラムダ変数の個数に、セットされる。

関数 `loader[x]` の引数は、関数 `compile[x]` の値と、同じ構文に、従う。従って、`loader[compile[x]]` と、使用したり、コンパイルした結果を、一時 `fprint[compile[x]; fileno]` で、ディスクに、格納し、後に、`loader[fread[fileno]]` で、ロードすることが、出来る。

表-2 変換規則

(X^* は、 X を変換した結果を示す。)

$$1. (\text{LAMBDA } v \text{ list } x_1 x_2 \dots x_n) \Rightarrow x_1^* x_2^* \dots x_n^* \text{RT}(n)$$

n : v -list の長さ

$$2. X \text{ が Nil, } \perp, \text{ 数値} \Rightarrow \text{LI}(x) \text{ or PLI}(x)$$

$$3. X \text{ が変数} \Rightarrow \begin{cases} \text{L}(d) \text{ or PL}(d) & : X \text{ がラムダ又はプログ変数} \\ \text{LG}(x) \text{ or PLG}(x) & : X \text{ がグローバル変数} \end{cases}$$

$$4. (\text{QUOTE } x) \Rightarrow \text{LI}(x) \text{ or PLI}(x)$$

$$5. (\text{COND } (P_1 X_1)(P_2 X_2) \dots (P_n X_n)) \Rightarrow$$

$$P_1^* \text{NJ}(\ell_1) X_1^* \text{J}(\ell_0) P_2^* \dots P_n^* \text{NJ}(\ell_n) X_n^* \text{J}(\ell_0) \text{LI}(\text{Nil})$$

↑

ただし $P_n \equiv \perp$ のときは、

$$P_1^* \text{NJ}(\ell_1) X_1^* \text{J}(\ell_0) P_2^* \dots P_{n-1}^* \text{NJ}(\ell_{n-1}) X_{n-1}^* \text{J}(\ell_0) X_n^*$$

↑

$$6. (\text{AND } X_1 X_2 \dots X_n) \Rightarrow X_1^* \text{NJ}(\ell) X_2^* \dots X_n^* \text{LI}(\perp) \text{J}(\ell') \text{LI}(\text{Nil})$$

↑

$$7. (\text{OR } X_1 X_2 \dots X_n) \Rightarrow X_1^* \text{TJ}(\ell) X_2^* \dots X_n^* \text{LI}(\text{Nil}) \text{J}(\ell) \text{LI}(\perp)$$

↑

$$8. (\text{PROG } v \text{ list } x_1 x_2 \dots x_n) \Rightarrow \text{BIND}(n) X_1^* X_2^* \dots X_n^* \text{LI}(\text{Nil}) \text{RT}(m)$$

n, m : プログ変数, ラムダ変数の個数

$$9. (\text{SETQ } X Y) \Rightarrow \begin{cases} Y^* \text{ST}(d) & : X \text{ がラムダ又はプログ変数} \\ Y^* \text{STG}(x) & : X \text{ がグローバル変数} \end{cases}$$

$$10. (\text{GO } X) \Rightarrow \text{J}(\ell) ; (\text{RETURN } X) \Rightarrow X^* \text{RT}(n)$$

n : ラムダ変数の個数

$$11. (\text{fn } a_1 a_2 \dots a_n) \Rightarrow$$

$$\textcircled{1} \text{MK or PMK } a_1^* a_2^* \dots a_n^* \text{CN}(\text{fn}) : \text{fn が subr}^*$$

$$\textcircled{2} a_1^* a_2^* \dots a_n^* \begin{cases} \text{C}\phi(\text{fn}) \text{ or } \text{C}1(\text{fn}) & : \text{fn が } \phi, 1 \text{ 引数 subr} \\ \text{C}2(\text{fn}) & : \text{fn が } 2 \text{ 引数 subr} \\ \text{C}(\text{fn}) \text{ or } \text{PC}(\text{fn}) & : \text{その他} \end{cases}$$

§ 6. 性能評価

コンパイラ・システムの性能評価の目安として、インタプリタによる実行速度ならびに、コンパイルされたコードの実行速度を、LISP コンテストで、用いた、TPU のプログラムと、SORT のプログラムについて、測定した。この結果を、表-3, 4 に、示す。

表-3, 4 より、コンパイラ・システムの実行速度は、インタプリタの実行速度の、SORT で、約 5 倍、TPU で、4 倍弱である。これは、コンパイラ・システムの、仮想機械による実現を、考えると、ますます、満足のいくところである。

TPU のプログラムに対して、コンパイル時間は、約 90 秒 (コンパイラを、インタプリタで走らせた場合)、オブジェクトの大きさは、約 2.5 kW である。これは、ラムダ式が、フリー・セル中に、占める大きさの約半分である。

表-3 TPU の実行時間 (単位 秒)

問題	1	2	3	4	5	6	7	8	9
インタプリタ	215.5	606.3	256.5	352.5	40.2	1041.9	229.1	158.2	119.7
コンパイラ	58.9	170.7	71.0	98.8	9.4	293.2	62.2	41.9	31.6
比	3.7	3.6	3.6	3.6	4.2	3.6	3.7	3.8	3.8

表-4 SORT の実行時間 (単位 秒)

長さ	20	40	60	80	100
インタプリタ	3.3	12.1	19.8	30.0	41.5
コンパイラ	0.7	2.5	4.1	6.3	8.9
比	4.7	4.7	4.8	4.8	4.6

表-5 静的出現頻度と動的実行頻度

命令	静的出現頻度	動的実行頻度
LI	44 (3.5)	15,025 (0.7)
PLI	21 (1.7)	7,909 (0.4)
L	236 (18.9)	455,426 (21.0)
PL	160 (12.6)	428,293 (19.7)
LG	2 (0.2)	3,584 (0.2)
PLG	7 (0.6)	8,623 (0.4)
J	99 (7.8)	93,896 (4.3)
NJ	77 (6.0)	274,296 (12.6)
TJ	10 (0.8)	23,930 (1.1)
ST	177 (13.9)	55,319 (2.5)
STG	3 (0.2)	3 (0.0)
MK	33 (2.6)	1,057 (0.2)
PMK	4 (0.3)	55 (0.0)
BIND	13 (1.0)	1,296 (0.3)
C	0 (0.0)	0 (0.0)
PC	39 (3.0)	18,390 (5.9)
Cφ	0 (0.0)	0 (0.0)
C1	214 (16.9)	52,442 (15.8)
C2	55 (4.3)	28,638 (8.9)
CN	37 (2.9)	1,112 (0.2)
RT	39 (3.1)	18,391 (5.9)

次に、命令が、プログラムの中に、どのような頻度で、現われるか (静的出現頻度)、ならびに、どのような割合で、実行されるか (動的実行頻度) を、測定した。その結果を、表-5 に、示す。ここで、動的実行頻度の測定には、TPU の問題 6 を用いた。

表-5 より、(1) L 命令と、PL 命令は、ほぼ同じ割合で実行されている。(すなわち変数のロードの半分は、レジスタへの、ロードだけで済み、スタック操作を、併ねない。)(2) コール命令では、C1 命令が、いちばんよく使われている。(この命令も、スタック操作を、併ねず、レジスタだけで、処理が、済む。)、ことが、わかる。

したがって、今回、採用した、仮想機械の構成は、実際に、妥当なものであると、思われる。

() 内は、%。

§ 7. あとがき

出来るだけ、高級言語で、システム記述を行なうという立場から、仮想機械によるLISPコンパイラ・システムの実現を、行なった。インタプリタと比較して、4~5倍の、処理速度の改善は、この立場から、まずまずの結果である。

現在、LISP38は、構文的文字認識の研究への応用に、学生のリスト処理の演習に、利用されている。今回、報告した、LISP50は、OKITAC system 50/40のDOSが、FORTRANユーザーに、48kwのメモリしか、使用させないので、主に、LISPシステムの開発の為に、用いられている。

今後の課題としては、コンパイラ・システムの整備を、行なうことと、LISP、PASCAL等の実現に、用いられている、スタック機械の、効率のよい実現、そのオブジェクトの最適化の研究を、あげられる。

謝辞

曰頃、御指導頂く、名古屋大学工学部 本多淑雄教授、三重大学工学部 大山口昭敏教授、また、熱心に、討論に、参加して頂いた、稲垣研究室、福村・本多研究室の諸氏に、深く、感謝致します。

参考文献

- ① 太田、吉田、福村、"会話型LISPの実現とそのGrammatical Inferenceへの応用", 記号処理研究会資料3-2 1978.
- ② 太田、吉田、稲垣、"会話型LISPシステム(LISP50)上でのコンパイラの作成", 電気関係学会東海支部連合大会 予稿 1979.
- ③ 太田、吉田、福村、"INTERLISPの機能を取り入れた会話型LISPの実現" 信学会総合全国大会 予稿 1977.
- ④ 47内、"LISP処理系コンテストの結果", 記号処理研究会資料5-3 1978.

付録 1. LISP 50 の組み込み関数

(fsubr)	(subr)	(utility)		
quote[x]	car[x]	rewind[x]	lessp[x,y]	edit[x]
setq[x,y]	cdr[x]	rplaca[x,y]	numberp[x]	pretty-print[x]
cond[x1;x2;...;xn]	cons[x,y]	rplacd[x,y]	member[x,y]	compile[x]
define[x1;x2;...;xn]	atom[x]	eval[x]	length[x]	call-tree[x]
and[x1;x2;...;xn]	eq[x,y]	apply[x,y]	reverse[x]	etc
or[x1;x2;...;xn]	equal[x,y]	gensym[]	set[x,y]	
time[x]	null[x]	minusp[x]	loader[x]	
prog[v;x1;x2;...;xn]	read[]	minus[x]	reclaim[]	
go[x]	print[x]	addl[x]	getd[x]	
return[x]	cread[]	subl[x]	putd[x,y]	
(subr*)	lprint[]	difference[x,y]		
list[x1;x2;...;xn]	eject[]	quotient[x,y]		
plus[x1;x2;...;xn]	fread[x]	remainder[x,y]		
times[x1;x2;...;xn]	fprint[x,y]	rand[x]		
append[x1;x2;...;xn]	prinl[x]	zerop[x]		
nconc[x1;x2;...;xn]	terpri[]	greaterp[x,y]		

付録 2. コンパイラ・オブジェクトの例

```

(FACT (LAMBDA (N)
  (COND ((ZEROP N) 1)
        (T (TIMES N (FACT (SUB1 N)))))
)

(FACT (N)
  (L . N)
  (C1 . ZEROP)
  (NJ . 10)
  (LI . 1)
  (J . 21)
  (MK)
  (PL . N)
  (PL . N)
  (C1 . SUB1)
  (PC . FACT)
  (CN . TIMES)
  (RT . 1))

(REV (LAMBDA (X)
  (PROG (Y)
    A (COND ((NULL X) (RETURN Y)))
    (SETQ Y (CONS (CAR X) Y))
    (SETQ X (CDR X))
    (GO A))
)

(REV (X)
  ((BIND . 1)
   (L . X)
   (C1 . NULL)
   (NJ . 12)
   (L . Y)
   (RT . 1)
   (L . X)
   (C1 . CAR)
   (PL . Y)
   (C2 . CONS)
   (ST . Y)
   (L . X)
   (C1 . CDR)
   (ST . X)
   (J . 2))

```