LISP マシンとその評価

LISP machine and its evaluation

島田俊夫　　山口喜教　　坂村　健

Toshio Shimada　Yoshinori Yamaguchi　Ken Sakamura

電子技術総合研究所　　　慶応大学

Electrotechnical Laboratory　　Keio University

1．まえがき

LISPは、人工知能の研究に広く用いられている言語である。LISPの特徴は動的記憶領域の割付や関数の再帰的呼出，リスト処理などであるが、殆んどの計算機はこれらの機能を効率良く実行する事ができない。実行時の処理を余り必要としないFORTRANのような高級言語では効率を上げるためにコンパイラが使用される。もちろんLISPのコンパイラも存在するが LISPはプログラムの中でプログラムを作り評価することがあるため、実行時にインタプリターが必要となり十分にコンパイルする事ができない。また、効率の良いコンパイラを製作する事は一般的に難かしい事であり，言語の機能を制約しがちである。

この問題を解決するためには 計算機の命令と高級言語の命令の間のギャップを縮め、高級言語を機械レベルでインタプリットする方法が効率の面からも、システム作成の面からも良い方法である。しかしこれをハードウェアで実現することには多くの困難が伴う。近年ファームウェアが発達しユーザーがマイクロプログラムを用いて新しい命令を定義する事が容易となった。本論文はこのような計算機の一つ*を用いてLISPマシンを製作し 高級言語のインタプリタ方式の評価を与えたものである。従来、高級言語計算機として提案されたものは多くあるがその大部分はペーパーマシンであり，実際に製作されたものもある特定の関数のみを実現したものが多い。そのためファームウェアを用いればシステム全体としてどの程度の性能が得られるかを知る手がかりとなるデータは殆んどない。我々はLISPに関してこのような総合的なデータを与える事を試みている。

また我々のLISPは人工知能の研究に用いることが目的であるので、最近の人工知能用言語、Micro-PLANNER，CONNIVER，QA4等を効率良く実行するために必要な機能として バックトラッキングとコルーチンを取り入れた。これらの機能を実現するためには Bobrowのスタックモデルが概念的に良くまとまっているが、このモデルは効率面で問題があるように思われる。そこでまずこのモデルをファームウェアで実現し――すなわちBobrowのスタックマシンを実現し――その上にLISPマシンを実現した。

2．Bobrowの制御構造モデル

この制御構造モデルは 人工知能用言語の制御に必要なバックトラッキングやコルーチンを表現するのに適しており，1つのスタック上に実現できる。スタッ

＊ ヒューレット・パッカード社 HP2100

クの制御は単なるLIFO（last in-first out）ではなく，Bobrowはこのスタックの操作を数個の基本関数を用いて記述している。このスタックを、LIFOスタックと区別するため、Bスタックと呼ぶ。dynamic storage allocationではなくスタックを用いる理由はdynamic storage allocationでは、記憶の割り付けに時間を費したり、一時的な変数に対して最大限の領域を確保しておかねばならぬ等，効率の点で問題が多い事による。

Bobrowのモデル（以下Bモデル）の概要を述べる。関数やプロセシージアの動的な環境を表現するframeというものを考える。frameの構造を図1に示す。frameは、引数名とその値をストアするbasic frameと制御情報や一時的な値をストアするframe-extensionに分けられる。frame extension内の制御情報で重要なものは変数の結合状況を示すA link（access link）と、関数の呼びの状況を示すC link（control link）ならびに、basic frameとの結合を示すB link（binding link）である。

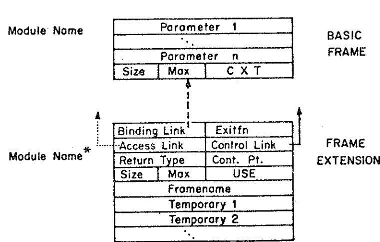


図1 frameの構造（文献1より引用）

バックトラッキングやコルーチンを実現するためには、あるプロシージアから他のプロシージアに実行の制御を移した後でも以前のプロシージアに対する環境（frame）を保存しておく必要がある。そのためBモデルでは特別のデータ型としてed（environment descriptor）を設け，これによって指されるframeは制御が他のframeに対応するプロシージアに移ってもスタック上に残り続ける。この事を例で説明する。今P1というルーチンが、P2を呼びP1がENVIRON(1)（命令表,表5参照）を呼んだ場合を考える。その時の制御はENVIRONのframeにあり、スタック構造は図2の如くである。ここで*はframe extensionを示す。CXTはbasic frameがframe extensionからB linkによって指されている数を表わし、USEはframe extensionが他のframe extensionやedによって指されている数を表わす。ENVIRONの実行終了に際して、EXAMという命令が実行される。通常この命令はENVIRONに対するframeをスタック・アップし、C linkで示されるframeに対して制御を戻すのであるが、この場合のようにUSEの値が2以上であると、そのframeのコピーを作り、コピーされたframeで実行を再開する。即ち図3のようになる。関数P2が実行終了してもスタックアップされるのは、コピーされたP2*のみで、オリジナルなP2*は、保存されていることがわかる。

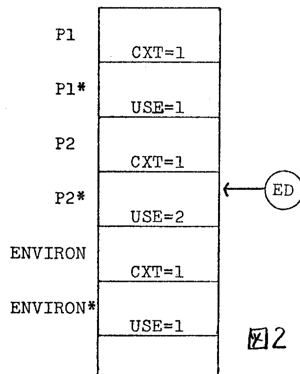


図2

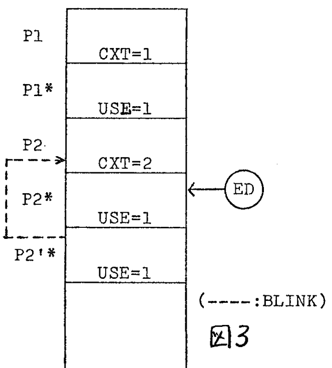


図3

図2 ENVIRON(1)実行中のスタック内部構造（文献1より引用）

図3 制御がP2へ移った時のスタック内部構造（文献1より引用）

3. BOBROW MACHINE

Bobrowマシン（以下Bマシン）とは、前節で述べたBモデルを性質として持つマシンである。Bモデル自体が言語の制御構造のモデルであるため、Bモデルを構成する基本関数を機械命令として持つようなマシンでのシステム作成は容易であり、作成された言語の効率向上が期待できる。

3-1 Bモデルとファームウエア

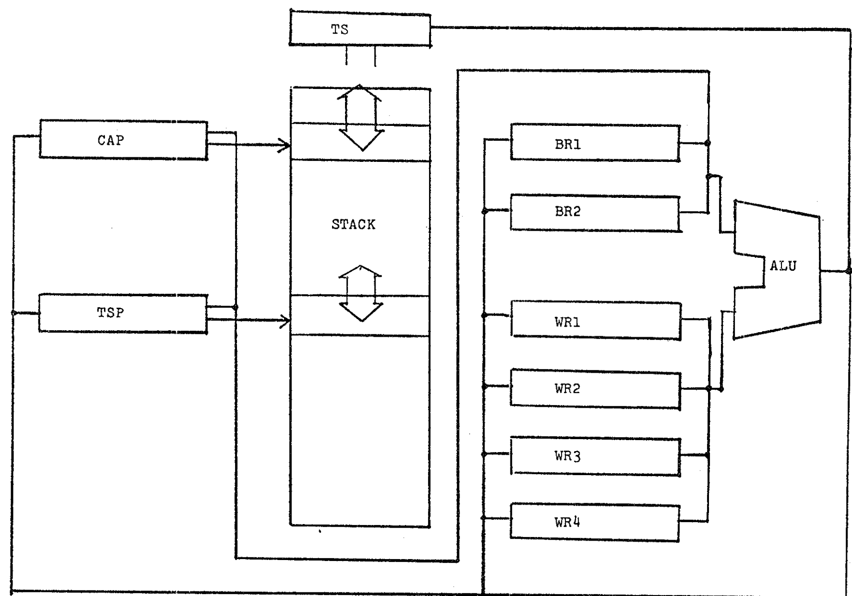
Bマシンの概念図を示そう（図4）Bマシンの中心となるのは スタック機構であり，このまわりに操作上必要となるレジスタ群（2つのスタックポインタ，2つの変数バインド用レジスタ，4つのworkingレジスタ）がならぶ．ALUの機能としては 加減算，比較機能，シフトなどが必要とされる。ファームウエアに用いるターゲットマシンHP2100は汎用マイクロプロセッサであるためレジスタやメモリ構成はこれとは異なる。ファームウエアにより，ターゲットマシンが、ユーザーにとって概念マシンと等価になるようにする。以下Bマシンについてスタック構造から述べる。

3-2 スタックの構造

LIFOスタックの基本概念は，一箇所だけからアクセスでき一番最初に入れたデータが最後に出てくるようなものであるが，Bスタックでは必ずしもそうではない。すなわちスタックへのアクセスがどこからでも自由にできpopしてもスタック内にデータを残しておける。普通のスタックのpush，popはデータ内容に関係なくどのように用いるかはユーザーまかせであるが，Bスタックではpushするのはframeという関数の実体であり frameの形式は関数ごとに変化するにもかかわらず 1個の命令で取扱える。

（1）中間領域へのアクセス

• スタックポインターとしてTSPとCAPを持つ事により任意の箇所へのアクセスを可能としている。TSPは常に現在使用中のスタックの底をあらわし、CAPがこれより下を指す事はない。（CAP≦TSP）TSPのみ使用するとLIFOスタックとなる。



CAP:Current Access Pointer; TSP:Top of Stack Pointer; TS:Top of Stack
BR:Binding register; WR:Working Register;

図4 Bobrowマシンの概念図

・LIFOスタックは出入口が1つしかないため原理的には中間領域からの読みだしもできない。Bモデルではたとえば A linkをたどっていってその関数が呼ばれた環境での変数の値を見るという事があるためpopをしないで、スタックの中を見る事ができる必要がある。

(2) スタック制御

スタックを制御する情報をスタック内データに付随させている。この事によりスタック制御の一部が、ユーザーレベルからマシンレベルに移り、システム作成は容易となった。

・push, popするデータ数制御　関数をスタックにframeという形でpushする時、変数の結合が必要となる。変数の数は、関数により異なり一定でない。push, pop命令ではユーザーが引数の数まで考慮してスタック操作しなければならないが、Bマシンではその必要はない。たとえばframeをpushする命令は、pushする関数の引数の数によらず1回だせばよい。

・edポインター　edリストから指されていると、スタック内データをpopしてもそのデータはスタック内にのこる。再びpushする時は、その上に再書きこみされないようスタックを操作する必要があるが、これらの処理もBマシンではユーザーが考慮する必要はない。

・スタックガーベージコレクション　edポインターがあるため、スタック各所に穴があく。ここをどのように使うかは状況に応じて異るわけで、この有効利用はシステムの効率向上にもつながる。又basic frameは、複数のframe extensionから指される事がありこのような場合にも、スタック内にbasic frameだけをのこすため穴があく。Bモデルでは穴のあいた所をミニスタックとして使用し、必要に応じてスタックの詰め直しを行うようになっている。

3-3 Bマシンの命令

Bマシンの命令決定に際して考慮した事は以下の通りである。

・原則としてBモデルのプリミティブ（Bobrowが提案したもの）はすべてマイクロプログラム化する。

・そのほか、言語作成上必要となるもの、たとえばBモデル内での変数サーチなどもマイクロプログラム化する。

命令	機能
ENAM	ある関数を実行中に他の関数へ実行を移すものであり新しい関数に対するframeがスタック上につくられる。現在実行中だった関数はsuspendし制御は新しくつくられた関数のform bodyに移る。
EXAM	現在実行中の関数を終了させ、C linkで指されるframeへ制御を戻す。
ENVRN	あるformを指すenvironment descriptorをつくりその値をかえす。この命令によりformをスタック上に保存できる。
SETENV	environ discriptorの値をかえる。
EEVAL	指定した環境のもとでformをevaluateする。

QLOAD,QSTORE	CAPの制御ができる。
PPUSH,PPOP	TSPの制御ができる。
REFRA	frameの解放を行う。
COPY	frameのコピーを行う。
DELETE	frameを消滅させる。

表1 Bobowマシンの命令

4. LISP MACHINE

バックトラッキング、コルーチンなど高度な機能をもつLISPプロセッサをBマシン上にLISPマシンとして実現した。その現状と実現方法を述べる。

4-1 新しい機能を持つLISP

Bobrowのモデルは、コンパイラを基本として作られており、関数の実行に対してframeが作られるが、インタプリタの場合も各関数の実行時にframeが形成される。但しEXPR[8]やFEXPR[8]の引数の結合は、BINDという関数に対する引数の結合として処理される。例えば階乗の計算をする場合を考えてみる。定義式を評価した後ではFACTORIALというアトムの性質リスト中に、次のリストがインジケータEXPRの下に登録される。

```
(LAMBDA (N) (COND ((ZEROP N) 1)
                  (T (TIMES N (FACTORIAL (SUB1 N)))) ))
```

次に、(FACTORIAL 3)を評価した場合EVALの引数に(FACTORIAL 3)が結合され、EVALの実行が始まる。前述した様に、評価関数群の動的状態はスタック上のframeとして表わされ、このときのスタックの内部状態は図5の様になる。LISP1.5[8]の変数リストはこの方式ではスタック上に置かれる事になる。FACTORIALのようにリカーシブのみの構造を持つ演算においては、Bスタック上にLISPを実現した利点はない。むしろA link, C link, USE, CXT等の制御情報をスタックにpushしなければならないため、時間的にも空間的にも、オーバーヘッドが大である。しかし人工知能に用いられるような非階層的な構造を持った問題の解法に対しては、柔軟な制御構造をLISPシステム自体が持っている事が効率面でも有利であると考えられ、BobrowマシンのLISPシステムの特長となっている。ここではその柔軟な制御構造の一例としてバックトラッキングを取り上げる。

バックトラックの実現のため2つのLISP関数を定義する。

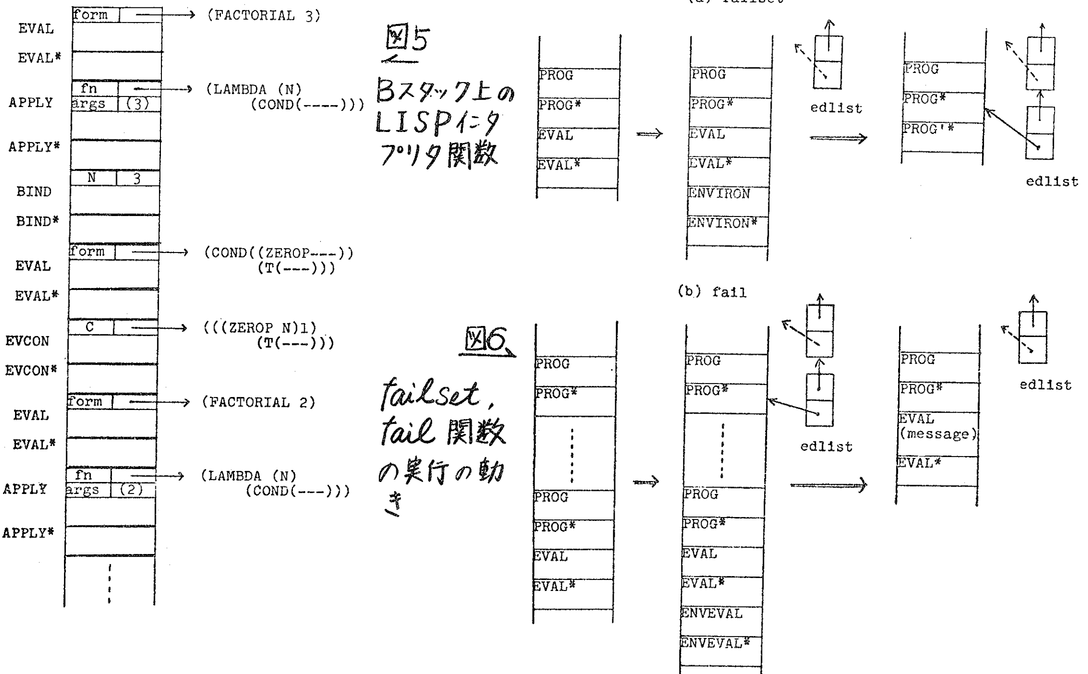
(1) (FAILSET)　この関数は、現在実行中の環境を、後でfail-backしてきたときのために保存しておく働きをする。

(2) (FAIL message)　この関数は、最も近いFAILSETされた環境で、messageという関数(又はプロシージア)を実行するものである。

これらを、Bマシンの基本命令を用いて書くと次のようになる。

```
failset()=prog( (); edlist←cons(edlist;environ(2)) )
fail(message)=prog((x); x←car(edlist);
                        edlist←cdr(edlist);
                        enveval(message;list(x)) )
```

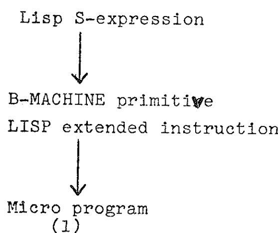
(注) edlistは、イニシャライズでNILにセットされるglobal変数である。
上の2つの関数の動きをBスタック上で図示したものを図6に示す。



Bマシンを使う事により fail や failset といった新しい関数をLISPに容易に導入できる事を示した。このような関数は、非決定的アルゴリズムに有用である。「8人の女王」の問題を 本LISPで記述した例を付録に示す。この例では SELECT という関数の実行環境が新しい選択をするごとに保存され、失敗した時点で fail を実行すれば SELECT で別の選択をするようになっている。

4-2 インプリメントの方法

ここでは Bマシン上にどのように、LISPプロセッサをファームウェア化したかについて述べる。方法として最も効率向上が期待できるのは図7(2)に示すようなLISPインタプリタを直接マイクロプログラムで書き下す方法がある。



Lisp S-expression

↓

Micro program
(2)

しかし(2)はデザインの点からもWCS*の容量の点からも望ましくないので図7(1)に示す方法ーBマシンの命令と、LISP用拡張命令を用意してシステム作成を行うーをとる事にする。

図7 ファームウェアの方法

* Writable Control Storage

4-3 拡張命令の決定とそのレベル

拡張命令のレベルを決めることは重要な問題である。言語設計を簡単にするためには高級言語と1対1に対応する命令を作るのが最も良いと思われる。LISPインタプリタの場合はそれを構成する関数を拡張命令として持つ事になるが、効率の面からは必ずしも良いとは言えない。たとえば図8はeval関数の一部であるが、numberかと聞いて次にatomかと聞いている。しかしこれをさら

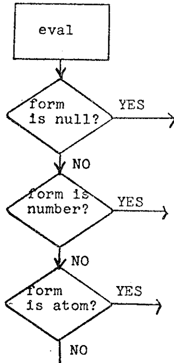
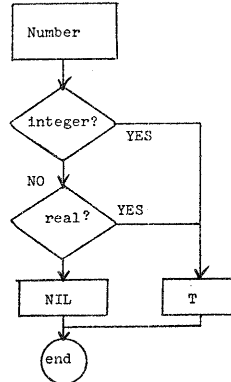


図8 eval関数の一部



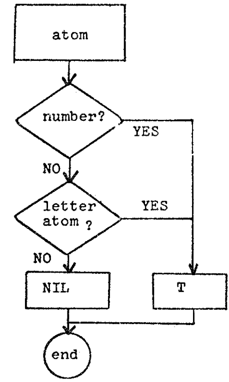


図9 atomとnumber

に分解すると、atomには、数値atomと文字atomがあるためatom関数の中で再びnumberかと聞いており同じ事を2度やる事になる。このような問題は、マクロ言語を使用する場合にも生じるが、マイクロプログラムで作られた命令は展開されるわけではないので、若干の相違がある。そこで以下の事を考える。

(1) atom命令をそれが使われる情況に応じて内容をかえていくつもつくる。

(2) atomよりさらに低レベルな命令をつくる。

(3) その部分をべた書きする。

(1)は、メモリ容量が許せば言語作成の面からも効率の面からも望ましいと思われる。(3)は効率面からみて最も良いが一箇所でしか使えぬというような命令の特殊化を招く。(2)はメモリ容量とか、命令使用度の平均化などからはすぐれていると思われるが、あまり命令を低レベル化すると、できるだけマイクロプログラムメモリ上で処理し命令のフェッチ回数を減らすというマイクロプログラムの利点がそこなわれる。我々の方針としては、可能な範囲で(1)をとり、部分的に(2)に従っている。そのため用意された命令(表2)には、CARが3通りあるという具合になっている。又(2)の低レベルの命令の決定は、実験をくりかえしながら決めた。たとえばLISPに多くでてくるリストをたどる部分だけを抜き出して1つの命令にしてみると、多くの箇所で使うので良いように思えるが、タイミングを考えるとサイクルの空きができて効率が悪くなるという事が起る。結局、拡張命令のレベルはマイクロプログラムの有効利用という点からは高い方がよいが、前述したシステム的な制約から高級言語と1対1に対応するわけにはいかないようである。またマイクロプログラムでは、メモリフェッチが少い方がタイミングの問題もなくなり効率も良くなるがそのためにはタグを利用したり、定数を高速メモリに置くなどの配慮が必要である。

命令	機能
CAR1 CAR2 CAR3	*(結果をだすレジスタが違う。)

CDR1 CDR2	＊（結果をだすレジスタの違い）
CONS	＊
EQ1 EQ2	＊
ATOM1 ATOM2	＊（命令の構成法の違い。）
NUMP	＊
GET	＊
FRGET	フリーリスト領域から1セルとってくる。
SLOAD	Bスタックでの変数サーチに使う。
SSTORE	Bスタックでの変数ストアに使う。

表2　LISPマシンの命令　　＊ 文献(8)参照

5。測定と評価

システムの効率を評価することは大変難かしい問題である。我々のLISPは新しい機能を備えているので単純に他のLISPと比較する事はできない。しかしながら，この新しい機能が従来のLISPより複雑な制御構造を持っており、これがオーバーヘッドになっていることは明らかである。まずオーバーヘッドをファームウエア化によって解消できるかどうかを調べる。

5-1 Bマシン評価

Bマシンにつくられたしたる LISPで、ハノイの塔のプログラムを実行した場合を図10に示す。HP2100のメモリサイクル時間はTOSBAC5600の2倍であり、実行時間も2倍だからこれを考慮すればほぼ同等のパフォーマンスである。従って我々のLISPは現存するLISPと実行速度は変わらず B-モデルのオーバーヘッドはファームウエア化により解消していると言える＊ 次に使用したスタックの最大長を表3に示す。我々のLISPは他のLISPの数10倍のスタックを使用するが、これはBモデルの性質上やむをえない事である。

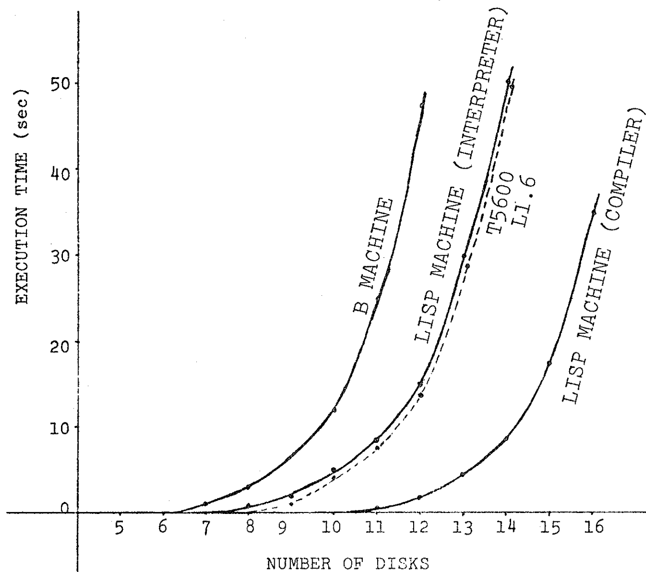


図10　ハノイの塔の実行時間の違い

	WANG	SORT	8 QUEEN	8 QUEEN (backtrack)
LISP-MACHIN	1920	9088	1408	2432
K-LISP	63	254	71	-
L-1.6	316	2536	270	-

表3　最大スタック長

(注：K-LISP　慶応義塾大学のLISP (T-3400)
L-1.6　TOSBAC 5600)

＊比較に使用したLISP1.6は、LISPコンクールにおいて他のLISPと同程度の性能を示している。

5-2 LISPマシンの評価

LISP拡張命令をeval関数の中に埋め込んだ時、実行時間がどの程度向上するかを%表示した。(表4) 又 速度向上率が全部のLISP拡張命令を入れたときの向上率に対しどのような割合を示すかも表にした(表5)。実行したプログラムは、タイプの異なるもの5つである。

Program / Extended instruction	Wang		Sort		8 queen	8 queen	Parsing
	A	B	20	60		(backtrack)	
SLOAD	32.8	33.4	31.2	31.2	30.8	30.2	30.4
SSTORE	14.1	14.2	14.8	14.9	13.5	13.7	13.2
GET	7.0	7.2	7.3	7.4	9.6	8.9	8.1
CAR	-	-	-	-	-	-	-
CDR	0.9	0.3	0.4	0.2	0.2	0.4	0.6
NUMP	11.6	11.2	11.0	10.7	11.0	10.5	11.0
FRGET	3.1	3.5	3.3	3.4	3.1	3.3	3.1

表4 拡張命令を埋め込んだ場合の実行時間上昇率(%)
(CARなどはCAR1~3までの合計)

(%)

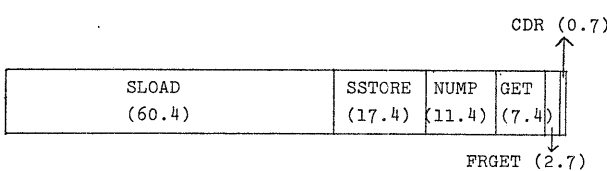


表5 LISP拡張命令すべて埋め込み上昇した分に対する各々の上昇率の割合

(1) どのプログラムでも速度向上率は、ほとんどかわらなかった。

(2) CAR, CDR のようなマイクロステップ数の少ない命令はほとんど効果がない。特にCARなどは、メモリリファレンスを1回するだけなので理論的にもHP2100機械語とスピードは同じである。マイクロプログラム化すると、ポインタ として使うレジスタと結果を入れるレジスタが種々のケースに応じ柔軟に制御できるよう命令をつくれるので、やや速くなる。CDRは、単独では アセンブラ の3倍であるが、このようにシステムにうめこまれた場合はそれほど効果はでない。

(3) もっとも効果の上ったSLOAD, SSTOREは Bスタックにおける変数のサーチを行うものである。探索する変数はどこにあるかわからないので見つかるまでA link をたどる。frameの中の変数の数やA link のたどりに応じて、簡単な加減算を行うのでマイクロプログラムの並列性が利用できソフトより数倍早い。又変数の探索では、利用できるレジスタ数が、マイクロプログラムの方が多く有利である。SLOAD, SSTOREは使用頻度も高くステップ数も多いので全体に及ぼす効果は大きい。

(4) GETはリストをたどるという点が SSTORE, SLOAD と異る。リストのたどりでは計算を必要としないのでマイクロプログラムの効果はSLOAD等にくらべて少ない。

(5) FRGETはフリーストレージ領域から1個セル(2ワード)をとってくる命令である。内容はポインタのつけかえでマイクロプログラムの場合ポインタをレジスタに置くことができるので、それだけ高速になる。

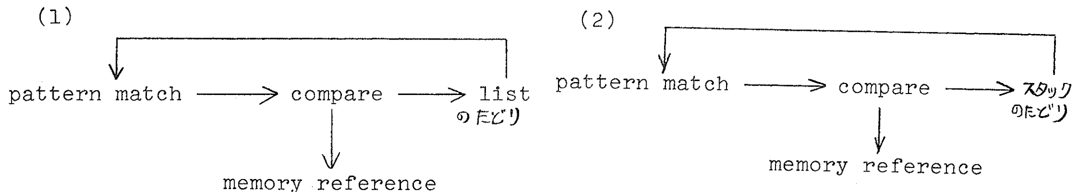
(6) 一般的にみてメモリリファレンス命令は大変多い。有利な点としては、メモリリードでは割合簡単にCPUフリーズのあき時間が使える事がある。メモリライトでもできるが、こちらの方がタイミングについて深く考えなければならないのでプログラミングは難しい。又これをまったく利用しないと、機械語と同等の速度になってしまう。

5-3 HP2100の評価と高級言語計算機に要求される機能

我々はHP2100を使う事により、Bマシン、Lマシンの実験を行なってきたがここではその経験に基づき（主としてマイクロプログラミングをした立場から）論じてみたい。まず作製した拡張命令の内容を調べ それとマシンの能力からくる制約との関係について考える。

マイクロコードの内容

今回作成したマイクロコードの代表的なタイプは次の2つである。



(1)にはGET、(2)にはSLOAD，SSTOREなどが属す。又その他の操作としては、複雑でない判断、比較が多い。複雑な数値計算の機能はLISPではあまり用いられない。

マイクロプロセッサに必要となる機能

(1) 強力な判断機能

マイクロプログラムにおいて最も並列的に動作可能であり、言語作成においても非常に多く用いられるので、高い機能がほしい。

HP2100ではいくつかのSKIP条件があるが そのほとんどが、HPの計算機のエミュレーションに都合の良いものばかりで、LISPには使えない。ここが、ユーザーが作れる任意の条件で判断SKIPできると都合がよい。

(2) マイクロプログラムレベルでのリカーシブコール

Bマシンの命令の1つEEVALでは、マイクロプログラムレベルでリカーシブな所がある。マイクロプログラム用スタックがほしい。

(3) マイクロプログラムレベルでのインダイレクト

リスト処理系の言語作成には特にほしい。

(4) イミディエート

LISP作成において、プロセッサ内で定数を使う事が多い。どのバスにも直接定数がだせると、マイクロプログラムステップが大幅に減る。

(5) ビット操作

これができると、スタック制御情報など1ワード内にもっとこまかく入れる事ができた。タグを取り入れるには必要な機能である。

6. 結び

新しいLISPの機能としてバックトラックとコルーチンは、ぜひとも必要な機能である。しかしそのインプリメントは難しく効率も悪いと考えられる。[4]
我々はBモデルをファームウエア化することにより、そのオーバーヘッドが殆んど無くなることを確めた。またLISPの一部をファームウエア化することにより相当の速度向上が期待できることもわかった。（アッセンブラーの5～6倍）

しかしながら、LISP全体をファームウェア化して評価することはハードウェアの制約上できなかったので、今後はBモデルを持たないLISPでこの点を追

追してみる。高級言語マシンにおけるタグの役割は、性能の面からも設計の面からも重要であるが我々の実験では16ビットマシンの制約からタグを採用する事ができなかった。これらの制約がなければ更に10倍程度の速度向上は可能であろう。高級言語マシンをファームウェアで実現した場合、何を持って成功とするかは難しい。性能の向上がなくても言語の作成が容易になれば十分な存在価値があるとも考えられる。我々の経験からは十分最適化されたソフトウェアの100倍の速度を得ることは難しく、10倍程度ではないかと思う。しかもこの点に関しては、良い計算機を使用したものに対しては同程度の性能しか期待できないという悲観的な意見(7)もあるので、他の実験の結果を待ちたいと思う。もしこの程度の性能しか期待できないとしたら、プログラミングの難しさもあるのでマイクロプログラムの目的をもっと他の所へ向けなければならないかもしれない。他の方々の意見をうかがいたいと思う。

謝辞

本研究の機会を与えて下さった慶応義塾大学の相磯秀夫教授、電総研の黒川一夫部長、御意見をいただいた飯塚肇氏ならびに計算機方式、推論研究室の皆様に感謝いたします。

参　考　文　献

1) D.G.Bobrow,"A Model and Stack Implementation of Multiple Environments", CACM,Vol.16,No.10,1973.
2) L.P.Deutsch,"A LISP Machine with Very Compact Programs",Proc.of 3rd IJCAI,1973.
3) R.L.Wigington,"A Machine Orranization for a General Purpose List Processor",IEEE,Trans on EC,Vol.EC-12,1963
4) 島田、山口、坂村　「LISPとLISPマシン」，昭和49年情報処理学会記号処理プログラミング・シンポジウム
5) 坂村、山口、島田，「人工知能用言語の制御構造に関する一実験」，昭和49年電子通信学会全国大会予稿集
6) 島田俊夫，「SLISPとLISPマシンの基本構想」，昭和49年電子通信学会全国大会予稿集
7) M.Barbacci,H.Goldberg,M.Knudsen,"C.ai--A LISP Processor for C.ai", Carnegie-Mellon Univ.Inner Report,1971.
8) J.McCarthy,et al.,"LISP 1.5 Programmer's Manual",MIT Press,1962.

付録

```
JOB KEN S 74
0001  (DE SELECT (SET UNDOLIST)
0002      (PROG (FLG SSET X)
0003          (SETQ SSET SET)
0004      SL  (COND ((NULL SSET)
0005                (FAIL UNDOLIST) ))
0006          (SETQ FLG NIL)
0007          (FAILSET NIL)
0008          (COND (FLG (GO SL)))
0009          (SETQ FLG T)
0010          (SETQ X (CAR SEET))
0011          (SETQ SSET (CDR SSET))
0012          (RETURN X) ))
0013
0014  (DE QUEENS NIL
0015     (PROG (N ANS M)
0016          (SETQ N O)
0017    LP    (SETQ N (PLUS N 1))
0018          (COND ((GREATERP N 8)
0019                   (PRINT (QUOTE (KOTAE DESUYO))) (PRINT ANS)
0020                   (RETURN ANS) ))
0021          (SETQ M (SELECT (QUOTE (1 2 3 4 5 6 7 8))
0022                       (QUOTE (PROG NIL (SETQ N (DIFFERENCE N 1))
0023                                    (SETQ ANS (CDR ANS ))))))
0024          (COND ((CONFLICT M ANS 1) (FAIL NIL) ))
0025          (SETQ ANS (CONS M ANS))
0026          (GO LP)))
0027
0028  (DE CONFLICT (CM CANS COUNT)
0029    (COND ((NULL CANS ) NIL )
0030          ((EQ (CAR CANS) CM ) T )
0031          ((EQ (CAR CANS) (PLUS CM COUNT))   T  )
0032          ((EQ (CAR CANS) (DIFFERENCE CM COUNT))  T  )
0033          ((NULL (CDR CANS))   NIL  )
0034          ( T  (CONFLICT CM (CDR CANS) (PLUS COUNT 1)) )
0035   ))
```

バックトラッキング関数（FAIL，FAILSET）を用いて「8人の女王」の問題を、Lマシンで解いた例。