Lisp マシン製作奮戦記 ①

井田昌之

1. マイコンでもかなりの仕事ができる

電源を入れる．いくつかの準備手続きを入力する．そこでタイプライタに向かって

```
(X+Y)**2;
```

と入力すると

```
 2        2
X +2*X*Y+Y
```

との印字がされる．

また $\frac{d}{dx}(x^2+x)-x$ を実行させようとするなら

```
DF(X**2+X,X)-X;
```

と入力すると

```
X+1
```

と整理された答えが返ってくる．

これは，筆者らがマイクロプロセッサ Intel 8080 を用いて作成した ALPS/I (Aoyama List Processing System/I) での実行の一例で，数式処理言語 Reduce* (本来なら大文字で REDUCE と書くべきところだが，全体的な見やすさのために Reduce と表わす) を走らせた結果である．

こうしたことを行なえるプログラムの実行には相当量の主記憶が必要とされたため，大型の計算機以外では動かなかった．ALPS/I は，マイクロプロセッサに大容量(256Kバイト)の記憶装置を接続することによってメモリ容量の問題を解決し，「さあ，マイコンでどこまでの処理速度が得られるか？」に挑戦したシステムということができよう．

本稿では，①ALPS/I とは何か？ ②ハードウェアを作った際の失敗談・苦心談，③システム・プログラムの作成の際の失敗談・苦心談などを，2回の連載で述べる．

ALPS/I は，記号処理言語 Lisp で書かれたプログラムを直接解釈実行(インタプリト)するように作られた専用システムである．Lisp 処理系とモニタは PROM (半固定記憶) に書き込まれているので，電源投入動作だけで Lisp 言語による入力を受けつけるようになっている．

ALPS/I の特徴はMITの CONS マシンのように Lisp 向きの特殊なハードウェアの製作ということではなく，マイクロコンピュータという拡張性の高いハードウェアの上にソフト的な努力を重ねてコンパクトに専用システムを構成している点にある．Y. Chu の分類 (in High level language Computer Architecture, Academic Press 1975) で考えると第2のタイプと第4のタイプの合いの子といえるマシンである．(その機能的な特徴などは情報処理学会誌に「マイクロプロセッサを用いた Lisp マシン：ALPS/I」(掲載待ち) としてまとめている．ここでは裏話に徹することにする．)

マイクロプロセッサを用いてこうしたシステムをなぜ作る気になったか？ それにはいくつかの理由がある．製作を思い立った約4年前(当時 8080は発表されたばかりで2万5千円ぐらいしていた)をふりかえってみると，比較的単純な発想に基づいていた気がする．それは『ゆっくりと誰にも邪魔されずに自分の仕事ができる専用システムがあったら……』という，おそらくは誰でも持つであろう気持ちからであった．

* ユタ大学で開発された言語．Lisp で書かれてあり，記号微分，多項式の展開，その他の数式の処理を目的としている．国内には東大後藤研究室の尽力により普及されつつある．

<例1>

```
EVALQUOTE ENTERED, ARGUMENTS...
BEGIN NIL
REDUCE 2....

(A-B-C-D)**4;
KEY*= *LPAR*
PROGRAM*= ((EXPT (DIFFERENCE (DIFFERENCE (DIFFERENCE A B) C) D) 4))
```

<例1の出力結果>

```
 4      3       3       3         2  2          2           2          2  2
A -4*A *B-4*A *C-4*A *D + 6*A *B  + 12*A *B*C + 12*A *B*D + 6*A *C

      2          2  2        3          2            2             2
 + 12*A *C*D + 6*A *D -4*A*B -12*A*B *C-12*A*B *D-12*A*B*C -24*A*B*

           2        3         2           2       3     4      3
C*D-12*A*B*D -4*A*C -12*A*C *D-12*A*C*D -4*A*D  + B  + 4*B *C + 4*B

3     2  2        2          2  2         3         2
 *D + 6*B *C  + 12*B *C*D + 6*B *D  + 4*B*C  + 12*B*C *D + 12*B*C*D

2         3     4       3         2  2       3    4
  + 4*B*D  + C  + 4*C *D + 6*C *D  + 4*C*D  + D
```

<例2>

```
(A+B+C)**4*5-DF((A+B+C)**5,A);
KEY*= *LPAR*
PROGRAM*= ((DIFFERENCE (TIMES (EXPT (PLUS A B C) 4) 5)
(DF (EXPT (PLUS A B C) 5) A)))
EVAL LIST= (QUOTE 0)
RESULT= (0)
```

<例2の出力結果>

```
0
```

<例3>

```
END;
KEY*= END
PROGRAM*= (NIL)
ENTERING LISP...

END OF EVALQUOTE, VALUE IS..
ENTERING LISP...        Lnspインタプリタに戻る.
```

図 1 ALPS-Reduce 実行例
下線部は使用者の入力を示す．KEY＊＝，PROGRAM＊＝ などはデバッグ用プリントである．いずれも使用者の入力終了後1分以内に答えが印刷される．

2年ほど前の Dr. Dobb's Journal (米国のマイコン雑誌) に次のような話があったと思う (何月号だったかは忘れてしまった)．

ある人が『マイコンに BASIC をのせて家で使っていたら，子供たちがそれを見て使いたくなった．仕方がないから各自が自由に使えるようにマルチユーザ BASIC*1 はないものだろうか』と書いたら，他の人が『そんな手間のかかるソフトウェアを作る必要はない．必要な個数だけ単一ユーザの BASIC が動くマイコン・システムを買えばよい．そのほうが確実でしかも安くつく』と書いた．(記憶が不確かで部分的に筆者の創作があるかもしれない．)

当時，この話を読んでわが意を得たりと思ったものだ．事実，システム作成の手間は比較的小さくて済むし，必要ならばハードウェアに細工・改良を自らの手で行なうことができる．また，システム・プログラムも自作できるので実行時のソフトウェア・オーバヘッドも少ない．他との会話が必要なら通信インターフェイスをつければよい．

こうした考えで ALPS/I は作成され，昭和51年1月から作動を始めている．その速度は大型機などの Lisp システムと比べて数十倍遅いが，システム・オーバーヘッドがないので，使用者の目から見た実効的な応答速度は桁違いに遅いというわけではない．たとえば理化学研究所の Hlisp-Reduce を TSS 端末から利用すると，システムのセットアップに約4分，$(x+y+z)^{10}$ の展開に約15秒を要する*2．ALPS/I はセットアップに約17分を要する．そして実行に5分，結果の印刷に6分10秒を要するが，一応実行できる．TSS 下での応答時間はもちろん不確定要素が多く，直接的な比較はできない．ALPS/I 上に実現された Reduce を ALPS-Reduce と呼ぶ．その実行例を図1に示しておく．

現在の ALPS-Reduce (初版) は約40Kを要し，約24K語の自由領域を持っている．仕様的には本来の Reduce の全機能はなく，サブセットになっている．フル機能を

*1 1台のコンピュータ上で複数人が並行してプログラミングおよび実行ができるようなBASIC．
*2 東大大型計算機センターの Hlisp-Reduce (コンパイラ版) では，この例の実行に2秒程度の応答時間 (CPU 時間で728ミリ秒：これは大変な速さである) しか要していない．

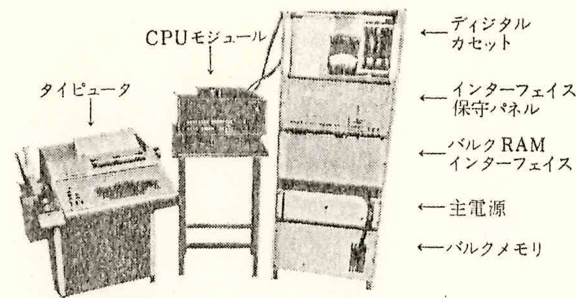

図 2 ALPS/I 外観

持つ第2版をのせようと現在も奮闘が続けられている.

2. Lisp には大きなメモリ空間が必要である

Lisp の紹介については他の記事, 文献にゆずることにするが, さるえらい先生をして「世の中には2種類の言語がある. すなわち Lisp とそれ以外である」と言わしめたものである.

Lisp は, 先に示したような数式処理やその他の大規模な人工知能関連プログラムを他の言語による場合に比べて, 容易にそして簡潔に書ける特質がある. そしてこれらの人工知能用言語や数式処理言語を移植 (他のマシンに移すこと) するには数百Kバイトが少なくとも必要であった. 本章の表題はそうした事柄をまとめて表わしており,「大規模な応用システムの実行をするには, 大規模なメモリが必要である」という, ある意味で自明なことを表わしている. そのほか Lisp に関する諸々の話は本稿の目的ではないので, 別の機会に譲ることにしたい. あくまで大型の仕事を走らせるためのマイコン・システムの話として扱うことにする.

ALPS/I の外観を図2に示す.

左にあるのがタイピュータである. 使用者はタイピュータに向かって直接 Lisp による入力を行なう. もし紙テープでプログラムを用意するなら, 紙テープリーダにセットするだけで自動的に読み込まれる. まん中に見えるのが CPU モジュールである. この箱 (筐体) は開発当初 (昭和49年2月) にただ一つ存在したマイコンで, インテル8080を中心とした CPU ボード (CPU チップ周辺のロジックはすべて TTL で組んである), 簡単なモニタの書き込まれた PROM ボード, そして RAM

*1 CPU チップおよびメモリなどの周辺チップがつけられたもの.

*2 メモリマップ I/O：メモリ空間の中に外部装置とのインターフェイスを埋め込んで, メモリ関係の命令を用いて入出力を行なう方法. 一般のポート接続に比べて簡潔さと高速性が得られる場合が多い.

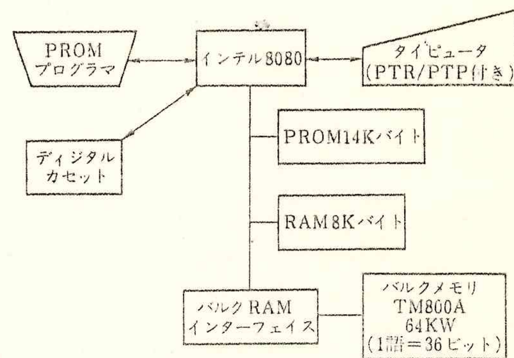

図 3 ハードウェア構成

ボード, I/O ボード, フロントパネル, 1702PROM 書込み器がつけられていた. エバリュエーション・ボード*1 とかキットなどは当時はまったく存在していなかったし, CPU 周辺チップもまったく存在しなかったので, 筆者らはこれを核として ALPS/I の CPU モジュールを構成した. そのため筐体的にはさまざまな修正が行なわれているが, これについては後述する.

図2の右に見えるラックには下から順に, 主記憶となるバルクメモリ (大容量メモリ), 電源部, バルクインターフェイス部, そしてディジタル・カセット装置およびそのインターフェイスが入れられている. バルクメモリ本体以外は自作である.

図3にそれらをまとめておく. なお現在, フロッピー・ディスク・インターフェイスが完成しており, Lisp とのソフト・インターフェイスを作成中である.

この構成の特徴は次のようにまとめることができる.

❶ メモリ全体を2セグメントとして構成する. 一方をインターナルメモリ, 他方をバルクメモリと呼ぶ. インターナルメモリはバイトアクセスされ, 64K空間を持つ. インターナル系は8080の機械語によって直接アクセスされ, またシステム・プログラムおよびその作業域がとられる. これに対しバルクメモリは Lisp データを保持するための64K空間を持ち, 語 (セル) 単位にアドレスされる. それぞれのアドレス情報は16ビットで扱われる. セグメントの区別はシステムの文脈中で行なう.

❷ バルクメモリ参照用の命令の追加. Lisp データ (バルク上のデータ) はバイト単位ではなく, その意味的な単位であるセルに対応して語単位にアドレスできるようになっている. しかし8080にはこうした語処理を行なう命令は存在しないので, その追加を行なった. 追加された特殊命令は, 簡易型 (高速型) メモリマップ I/O*2 とでも呼ぶべきもので, バルク1語とインターナル系内の RAM 5バイトのあいだの転送を制御する.

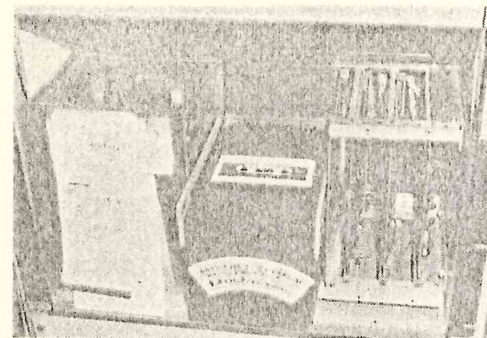

図 4 ディジタル・カセット装置

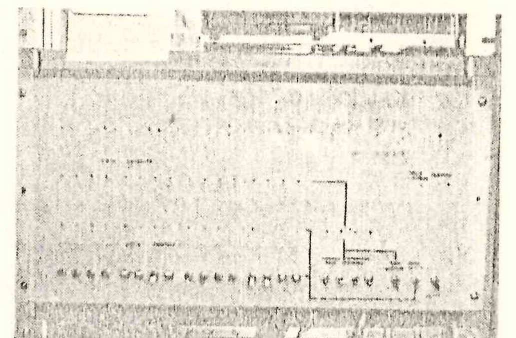

図 5 インターフェイス保守パネル (LED は秋葉原で安売りしているもので, 若干 "ガチャ目" である)

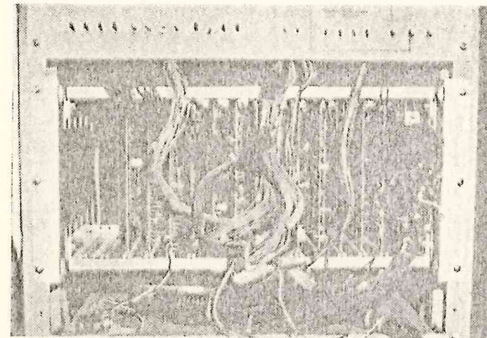

図 6 バルク-RAM インターフェイス (左のすきまには一時期おまもりがはいっていた)

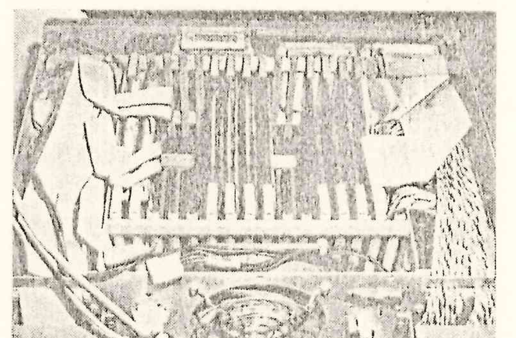

図 7 CPU モジュール内 (このうち7割程度はメーカ製. 中央の4枚は PROM ボード)

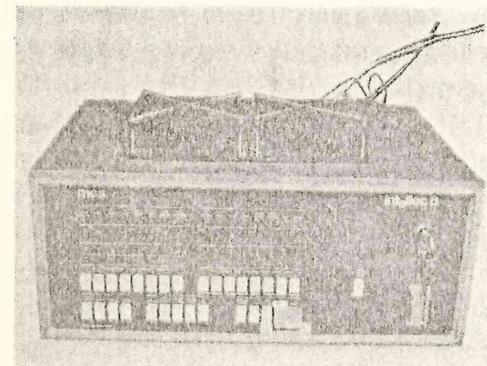

図 8 CPU モジュール外観 (右上へのびるコードは電源用. 上板はダンボール使用. 現在は 2716 ボードが作動したのでアルミの上板に変更. 右下ソケットは二代目になっている.

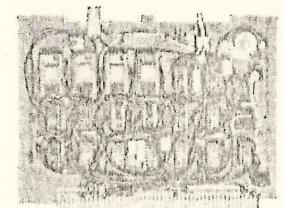

図 9 2716 PROMボード (2716 が4個のせられており 8Kバイトある. ソケットにさしたままで書込みができる)

❸ バルク上のデータの転送に際して RAM 側のバッファは, 16通り用意されている. バルク上のデータは, RAM 上のバッファを通して処理を受けるが, Lisp の特質から並行参照が必要な (あるいは並行して参照するほうが得な) 場合は少なく, バッファ数は16通りで充分である (リストたどりが多いなど).

❹ 自作した回路はこのバルク-RAM インターフェイスを中心として電源部, ディジタルカセット・コントローラおよびインターフェイス, 2716PROM ボードなど. 購入後に修正や大幅な改良・保守作業を行なったものにはタイピュータ (番号から見ると生産開始直後の49号機のようだ) とそのインターフェイス, CPU 筐体とその内部など, 購入時のままの使用もしくはメーカ側の保守によるのは, バルクメモリ本体, RAM ボード, 1702 PROM ボードなどである. 図4にディジタルカセット

表 1　ALPS/I 開発経過

年月	経過
1975年 2月	Intellec 8 Mod 80 を扱いはじめる
3月	タイピュータのインターフェイス作成
4月	TTY インターフェイス修正プログラムの PROM 書き
5月〜7月	バルク・インターフェイスのデザイン
8月	バルク・インターフェイスの図面作成 タイピュータのインターフェイス・ハードの虫取り Lisp インタプリタのコーディング
9月〜	バルク・インターフェイスの作成
10月	シミュレータとアセンブラのコーディング
11月	クロスシステム上で Lisp システムのデバッグ
12月〜	システム・テスト
1976年 1月	PROM 書き．初版の作動開始
5月	再システム・ロード バルクメモリのビット欠ける
8月	Lisp インタプリタ新版のコーディング ディジタルカセット・インターフェイスの作成開始
10月	新バージョンの PROM へのロード （この後たびたび行なわれる） バルクメモリのダウン
11月	バルクメモリのダウン 電源部改造
1977年 5月	システム・デバッグと PROM のリロード （インテルのチップに戻した）
8月〜9月	テスト・プログラム（箱入娘パズル）でシステムの虫の検出．タイピュータのダウン．バルクメモリのダウン．浮動小数点パッケージ作成
10月〜	システム・バグの検出と PROM の書直し．タイピュータのダウン．バルク・インターフェイスの改良
1978年 1月	バルクメモリのダウン．PROM ライタ不良． 空調ファンの取りつけ ディジタル回路シミュレータと Reduce のインプリメント
2月〜	2716 PROM ボードの作成
3月	フロッピー・インターフェイスのデザイン
4月	2716 ボードの採用

装置，図5，図6にバルク-RAM インターフェイス，図7，図8に CPU モジュール，図9に 2716 PROM ボードを示す．

3.　図面ができてもハードウェアは動かない

前章ではハードウェア構成の概略を述べた．

ここでは現在にいたるまでの開発・保守で悪戦苦闘した様子を記憶をたどりながらまとめてみよう．

まったく標題のとおりで，ハードに関しては素人だった筆者は，何度も何度も失敗を重ねた．設計の正しさ，実装技術の高さ，そして保守に対する根気と執着心の強さは，ソフト以上のものが必要だと感じている．

表1に恥をしのんでいままでの経過をまとめてある．

まず，1974年当時のマイコンをとりまく環境から始めよう．

8080 と 6800 が存在していた．しかし 6800 はまだ発表されて日も浅く，エバリュエーション・ボードなども当然存在せず，また周辺チップも少なく，使えるような状態とはいいがたかった．それに比べて 8080 は 4004 以来の蓄積からか，すでに PROM プログラマもつけられたシステム開発用モジュールまで市販されていたのである．この状況の中で両者の命令セットなどを比較しても，あえて 6800 を用いるほどに 6800 が優れているとはいえなかった*1．

しかし 8080 も現在とは違って，バスや外部信号の同期などに便利な石はなかったので，まったくの最初から作るのでは目的まで到達できそうに思えず，Intellec 8 Mod 80 というシステム開発用モジュールを核として開始することになった．

Intellec 8 Mod 80 には，PROM 書込み器（最近調子が悪い*2），各 4 つの入出力ポート，CPU ボード，デバッグ用モニタ（Lisp システムの開発にはほとんど役に立たなかった）PROM，RAM 8K バイト，フロントパネル・コントローラがついていた．

インクジェットプリンタ・インターフェイスは Intellec 8 についている 20 mA カレントループ・インターフェイス*3 に対して，−9 V ではなく 0 V に一部のロジックを変えることにより比較的容易に接続できた（半日程度）．これは 110 ビット/秒であったため，気軽な気持ちで高速化を試み，2400 ビット/秒にすぐにあげてみると，案の定，転送エラーになり動かなくなってしまった．症状を調べてもわからないので，仕方なくソフトで無駄なループを行なってもとの速さで動かそうということになった．そしてモニタ中のタイプライタ・インターフェイスを書き換えようとしてはじめて，プログラムが PROM に入っているからそう簡単には変更できないのだという事実に直面したのである．

*1 レジスタ構成および命令構成を考えると，やはり8080のほうに分があるといまでも思う．

*2 年をとったのかもしれないが，研究室の一人がプログラマ回路に使われているトランジスタの定格がよくないことに気づき，別のものと取り換えてくれた．その後はまあまあよいようである．

*3 通常のテレタイプ・インターフェイス．

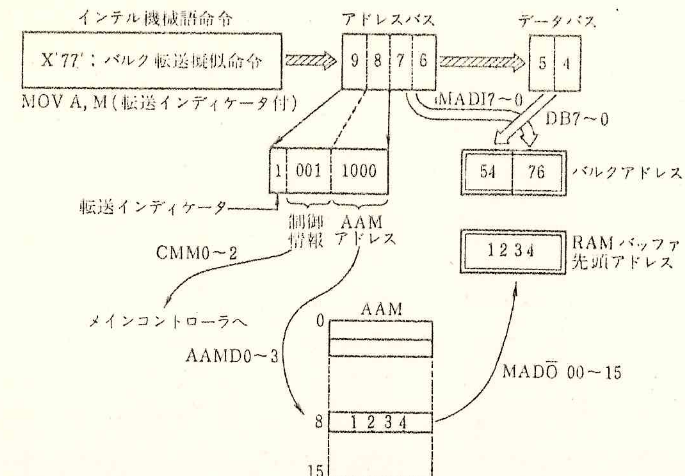

図 10　転送命令からのバルクアドレス，RAM アドレス，転送制御情報の生成

10ステップぐらいの追加がけっきょく 2 日かかった．まず秋葉原へ PROM を買いに行き，PROM プログラマの使い方をにぎやかに議論しながらおそるおそる書込みをしたこと，そして焼けた PROM（プログラムすることをいつの頃からか「焼く」という言葉を使うようになった．ソケットから「焼きたて」の PROM をとりはずし，「あちっ」といいながら手の中をころがし，次々と処理する様子はさながら，ふかしたての饅頭を食べるときのようである）を装着して動いたときの喜びは，いまでも思い出される．

PROM はその後も何百回と焼いたが，この最初の 1 個はすぐに不要になった．タイプライタ・インターフェイスに不具合が見つかったからだ．UART のビジーの見方がテレタイプ専用だったのである*1．

110ビット/秒なら問題なかったのだが，2400 ビット/秒にあげたらオーバーランを起こしてしまい，考え込んでしまった．人の作った部分はソフトもハードも一応疑ってかからねば大変なことになる．たとえメーカが作ったものであっても．（幸か不幸か筆者自身がハンダを握った部分にハンダ不良がたくさん出て，後輩からその点について非難を浴びた．人のことはいえません．）

操作卓になるタイピュータがついたので，次にバルクメモリの接続へと進む．（大容量のメモリをつけるのが筆者らのねらいだった．）

ここで問題になったのは，速さをできるだけ損わずに，安くそしてわれわれの手で作成可能な接続形式を見つけることだった．

まず入出力ポート接続を考えた．この方法は一番簡単に思えたが，速度が非常に遅い（1 語アクセスに 100 μ秒以上要する）．また，できればメモリアクセスのビット幅を大きくして語単位で扱うほうが速度，アドレス空間の節約に好ましい点を考え，考慮対象からはまっ先にはずした．

したがって，バルク 1 語と RAM 複数バイト間で転送を行なう DMA（ダイレクト・メモリ・アクセス）インターフェイスに目が向けられたわけである．いくつかの方式が考えられたが，最終的に決まったのは CPU で直接アドレスできる64K空間のうちの後半分はメモリ空間としてではなく，バルク・インターフェイスに対するコマンド空間として用い，その中に転送制御情報，RAM 側およびバルク側アドレス指定を凝縮させる方法であった．現在考えてもこれ以上の方法はないと確信している．（現在ハードウェア・セグメンテーション機構が IMSAI のマイコンにつけられるようになっているが，アドレス空間の切分け，速度*2 の点であまり良いとは思えない．）

インターフェイス時の情報の流れを図10に示す．現在ではようやく安定してデータ転送が行なわれており，エ

*1 UART の初段の ready しかみていなかったのである．ボードには余裕がないので，この修正は IC をさかさまにして接着剤でとめてある．

*2 バイトアクセスを行なうので，語すなわち n バイトを参照するには n 倍の時間を要してしまう．実効的に $1/n$ の速さのメモリ（たとえば 200 n 秒）を，ALPS/I では接続している（RAM 1 バイト＝1 μ秒，バルク 1 語＝1 μ秒）．インターフェイス・オーバーヘッドは ALPS/I 方式とセグメント・スイッチングとではあまり変わらない．

ラーは皆無といっていい状態になった．（1978年1月に行なわれた3人交代での100時間ぐらいの連続運転にも耐えられるものとなった．）

いままでに発見されたトラブルの原因はおよそ次のようなものである．

❶ハンダ不良

これは初期開発時によくでたものである．俗にいうテンプラハンダが結構あった．それはハンダ付けを行なった作業員の技量不足（特に筆者が行なった部分）によるものがほとんどだった．該当する部分は完全に最初から配線しなおすことにより解決した．

❷実装ミス

主に未熟さに起因する面が多かったと思うが，ラッチ（データの取込み）するクロック入力に対するファンアウトが不足していたり*，コネクタの不良，配線ミス，電源容量が足りなくなりCPUモジュールを含む全電源の見直しをしたことなど．

❸温度条件

CPU，RAM，PROM を入れてある筐体（Intellec 8 Mod 80）は16Kバイトまでしか扱えないよう設計されていた．これに対してマザーボードを修正し，20Kバイトを実装させ，さらにバスの外部取出し用のボード（バスエクステンダ）を入れた．このため，筐体内の温度が上昇し，動作不良を起こしたので，ファンを2つ秋葉原で買ってきてつけた．これによって動作時の PROM が指でさわれないくらい熱くなっていたのがほとんど平常に下がり安定した．図8のダンボール製の上板はその結果である．（4月になって2716を採用し，条件がゆるくなったので現在はアルミ上板にもどしている．）こう書けば実に単純な話で味気ない文章になってしまうが，このことがわかるまでに2年もかかった．初期のうちは小さな問題なら動くが，ちょっと長くなると（分のオーダになると）誤動作をしたりしたものである．

❹経年変化

操作卓となったタイピュータの故障が1977年の夏から1978年にかけてどっと現われた．インクを導くチューブの破損，紙テープリーダやパンチがおのおの独立に完全にエンコしてしまった（数回）．UART チップのピンの経年変化による接触不良．これらの修理およびトラブルシュートは間野研究室の後輩によって行なわれたが，締切におわれてプログラムをデバッグしている最中のダウンなど，怒る気力を失って黙々と修理したものである．

またバルクメモリに対してサイクルスタート（メモリサイクル起動信号）をかけるパルスは150n秒±50n秒という仕様がメーカから与えられていたが，今年の1月に思い出して見ると電源投入直後で120n秒，少し使ったら100n秒ぎりぎりしかないので，時定数をきめる抵抗を変えたりしたこともある．

そのほか，スイッチのへたり，発光ダイオード(LED)のつぶれ多少．

❺素子不良（?）

動作不良をシュートしていって原因がわかったとき一番ガックリしてくるのがこれだ．その最たるものは，国産某社の1702コンパチブルな PROM であった．初めてそのチップを手にしたときは Intel のものより安く，また足もがんじょうで国産他社のセカンドソースと異なり，書込み条件も同じなので，小踊りして喜んだものである．ただ紫外線消去用ののぞき窓からみえるウエハの大きさがひいき目にみても1ミリ角ぐらい大きかったのに気づいてはいたが……．このPROMにすべてのシステムをのせかえ，1976年11月に青山学院で記号処理研究委員会は開かれたのである．そのときまで動いていた Lisp サンプルプログラムが突如として動かなくなってしまった．さあ大変．けっきょく当日は冷や汗をかきながら泣く泣くさらに小さなサンプルを流して委員会の方々に見ていただいた．このくやしかったこと．

各自が思い思いのメモリテスト・プログラムを作ってテストしたが，それらではメモリはちゃんと動作するので，そのときの動作不良の原因はけっきょくわからなかった．

PROM チップに主な原因があることがわかったのは，翌年の5月．さらになんとこのとき使用したインテル社製のプリントボードの一部に不良があることがわかったのは実に12月になってからである．PROM チップの不良に気がついたのは偶然のことだった．システム・プログラムのデバッグ，機能拡張を行なって5月に PROM に書きなおすときに，なぜか国産某社のものではなく，その前に使っていたインテル社製のいくぶん貧弱な外観のするものをすべて用いたのである．何の気なしに電源をいれ，以前に正しく通ったプログラムをキーボードからたたいてみた．なんと動くではないか．驚き．この間ハードには何も手を入れていないのである．その後某社の PROM は何回か試みてみたが，やはり小さなプログラムなら動作し，数十秒から数百秒もランダムにアクセスされぐるぐる回る Lisp プロセッサに使うと，あらぬ動作をはじめる．これは素子不良ではないかもしれないが，賢明な読者の方にその真の原因をお教え願いたいものである．

素子不良としてもう一つのやっかいなものはやはりメモリだった．バルクメモリとして用いた国産T社の64K語についてである．ハードウェアの専門家でないので評価はできないが，全2.3Mビットのうち正味2年半のあいだに6ビットほど欠け，またドライブ回路の不良が3件ほどあった．

バルクメモリ関係のエラーがでると大変だ．こちらの作った回路が悪いのかもしれないし，場合によってはプログラムの虫が引き起こしたハードエラーもありうる．慎重に確認した上でメモリメーカT社を呼ばねばならない．これには8現象のロジック・アナライザが力を発揮した．single triggering 機構* を用いるとエラーが起きたときのただ1回の状況を最大8信号までストレージしてくれる．これが使えたので，あやしき部分に目星をつけ網をはってひたすら待つ．プログラムを流してお茶でも飲んでひたすら落ちるのを待つ．落ちたときにアナライザに駆けより症状がでていれば犯人を探し，でていなければ別の所に網をはることにしてまた最初からやりなおす．これで完全に犯人を見つけることができた．メモリ関係だけでなく原因不明になりそうなトラブルはすべてこれで解決した．ある特定のビットがふらふらしているといったケースはインターフェイス上に保守用パネル（図5）を作っておいたので，それをじっと見ているとと不安定なビットがぴかぴかとまたたきはじめる．これを目で確認してメーカの技術員を呼んだ．

T社のメモリはそうした点を除けば価格・技術員とのやりとりなどは比較的よかったと思う．念のため．

❻疲労による操作ミス

これは主に PROM の書込みに関する問題である．1702A は1チップに256バイト格納でき，紫外線を約20分照射することによって内容を消すことができる．256バイトの書込みには約3分かかるので，12Kバイトを書くのに全部で48個の PROM を使う．スムーズにいったとしても約150分要し，1回の書換えにおよそ3時間かかることになる．初期の頃はボードへのピンのさし方が悪く，ソケットに入っているつもりが，ピンを内側に曲げてしまい動作不良で困った．ピンが内側に曲がってしまっても外観だけでは最初はわからなく，またインテルの最初の頃の足は非常に弱かったのである（図11）．また，寝ぼけてくると PROM を別のソケットへさしてしまったり，誤って，できたばかりの PROM を消去してしまったりしたことを思い出す．

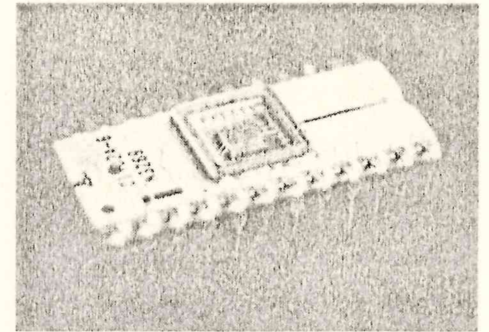

図 11　大活躍したインテル社製 1702A

現在8Kだけ使用している2716ボード（図9，現在最高の密度をもつチップ2716を実装）では，このことが嘘のような手順になっている．紙テープからオブジェクト・プログラムを読みながらこのモジュールへのメモリストア命令によって書込みを行ない，テープが読み終わると同時に完了する．これをオンボード・プログラミングという．（もちろん一般使用時に誤まってメモリストア命令が PROM に対して与えられても書込み動作をしないよう保護するマニュアル・スイッチが設けられている．）ほんの数年のあいだに大変な進歩があることを痛感している．しかし2716はまだまだ品不足で非常に入手が困難である．5V単一電源で動作し，1702の8倍の容量を持つこのチップが早く廉価になり広く出まわることを望んでいる．（入手にあたってご協力いただいた代理店の諸氏に感謝いたします．）現在チップ数が足りないので，8Kバイトを2716で，6Kバイトを1702で保持しているが，すべて2716になったなら，メモリのアクセスを現在の1μ秒から500ナノ秒に上げる予定である．

〔全2回〕　（いだ まさゆき　青山学院大学）

* 電子回路では，通常，電圧を信号として用いている（たとえば TTL では 0Vを0，5Vを1に対応）．そして，電流については特別気にせずに回路を組むことができる．しかし，電流があまりに少なければドライブできず，1つの素子の出力をあまり多くの素子の入力につなぐことはできない．たとえば，ファンアウト10というのは，10個の素子まではドライブできることを表わしている．しかしファンインが2という素子に対しては，5個以上ドライブできないことになるなどがあり，設計上注意がいる．

* 入力信号線のいずれかを特定して，その立ち上りないしは立ち下がりが1回起きたときに，他の7本の状況を記憶させ表示させるもの．

Lisp マシン製作奮戦記 ②

井田昌之

1. Lisp マシンをなぜ作ったか？

前回は，筆者らの作成した ALPS/I のハードウェアでの苦心談が中心であった．

その要旨は，次のようなものである．①専用機があると Lisp の実行に便利だという最初の動機，②Lisp プログラムの効果的な作成には，たとえば100以上の，多数の関数の組込みと大容量の主記憶が必要である，③日進月歩するハード，ソフト技術を取り入れられるように考える必要がある，④このため8ビットマイクロプロセッサを CPU チップとして採用し，これに大容量メモリを専用インターフェイスを通して接続する，⑤こうしてつくられた ALPS/I の作成維持は（筆者らが不慣れなためもあって）かなり大変であったが，一応の成果が得られた．

筆者らにとってこのマシンは，作るのではなく動かすことが目的である．このためハードウェアへの機能の組込みは，極力必要不可欠な部分だけにし，維持をなるべく容易にするように考えた．システム全体の機能性はソフトウェア上の努力に基づけようというわけである．逆に，まだまだアルゴリズム的にも機能的にも解決を要する問題が多く，本当にハードウェア化（ないしはファームウェア化）された Lisp マシンが出現するには，残念ながら多少時間がかかると思われる*．このことについての筆者の雑感を次に述べておく．

本年第2回の Lisp 処理系の性能コンテストが行なわれ，その結果が情報処理学会の記号処理研究会資料5に載せられている**．その結果と，1974年にまとめられた第1回のコンテストの結果とを比較してみると，それぞれの作成者の努力のあとがみられる．たとえば，東芝の黒川氏らの作られた電総研用の Lisp は，ある問題（Bit B-8）では10倍以上，平均的にみても数倍速くなっている．さらにコンパイラ版では数倍速くなっている．ほかの Lisp 処理系でも同様の高速化が認められる．これらはハードウェアの助けを借りない高速化である点に注目を要する．

また機能や仕様にしても，リストの内部表現の仕方について（cdr ポインタの削除・圧縮など），ガーベジ・コレクタについて（パラレル・ガーベジ・コレクタ，リアルタイム・ガーベジ・コレクタなど），ストリングの扱い，データタイプの導入，数値演算の高速化，多倍長数の導入，a-list を用いない束縛環境下での関数引数問題，そして各種関数の仕様の違いをどうつめるかなど，地道な作業と研究が必要である．ファームウェア化はこれらの結果をふまえて進むべきであると思う．その意味で ALPS/I は ALPS プロジェクトの一里塚といえ，アルプスのふもとにシャモニという町ができたところという感じである．この間の流れを図1にまとめてみた．

さて，以下に ALPS/I にのせられたソフトウェア開発の経過と，のせられたソフトウェアの中味の一部を述べる．多少なりとも読者の参考になれば幸いである．

* 実際的に有用な Lisp マシンは，筆者の知るかぎりでは，アメリカでも MIT のものだけのようである（bit 5月号，米澤氏の記事参照）．

** 武藤野通研の竹内氏の労によりまとめられた．筆者の ACPS/I を含めインタプリタ版17システム，コンパイラ版9システムが参加している．なお，本誌の86ページに竹内氏の解説がある．

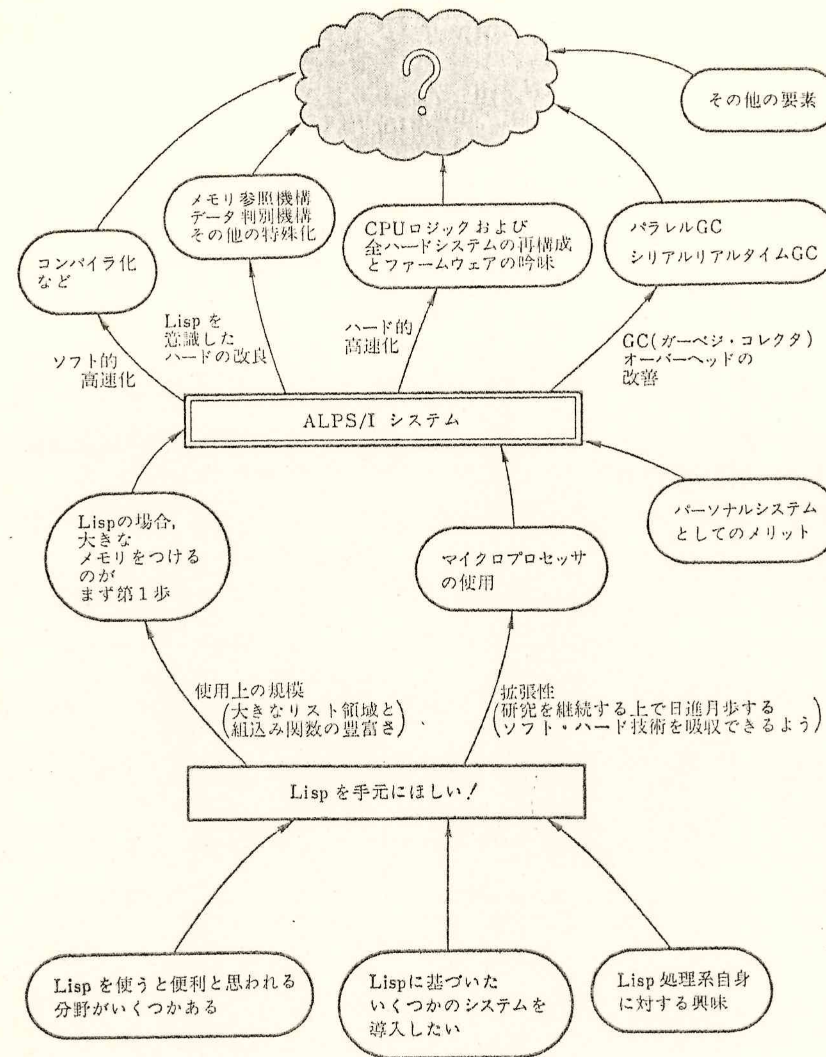

図1 ALPS システム開発の流れ

2. 良きクロスソフトウェアなしに仕事は始まらない

開発当初は，マイクロコンピュータという言葉さえ，ほとんど知られていなかった．したがって，一部に存在していたミニコン用以外には，クロスソフトウェアという概念はあまり知られいなかった．Lisp の処理系のコーディング総量は大きくなることが予想され，また修正・改良も多数回行なわれると予想された．このためハードウェア作成に並行して IBM 370/138 上に，アセンブラおよび ALPS/I シミュレータを作成した．

筆者は IBM のアセンブラに慣れていたので，当初は完全に IBM 型のアセンブラにしようと思った．しかし，他との互換性を考え（インテル形式のプログラムも通せるほうが望ましいので），マクロ定義も定数の取り方もインテル形式にし，シンボルとしては8字まで許すことと，コロンの有無を無視することだけを追加したアセンブラを後輩に作成してもらった．このアセンブラは PL/I（最適化機能付き）を用いて書かれており，カード約8000枚のソースプログラム（17個の使用者マクロ定義含む）を約20分でアセンブルする．

アセンブラ開発の経過は自転車操業的であった（時間も不足し，マンパワーも不足していたので）．Lisp 処理系の初版は75年夏に約3週間でコーディングをしてしまった（約5000枚）．この時点ではアセンブラは存在していない．カードをダンプしたリストを見て机上デバッグを行なった．アセンブラおよびシミュレータは，その後2カ月ぐらいで完成した．アセンブリ・リストの例を，

```
                  01727 ;  **********  EVCON  *** ①
                  01728 ;
09A0              01729 EVCON   EQU   $
09A0    2ADB3F    01730          LHLD  ZZADR1 ②
09A3              01731          TVSAVE        ③
09A6              01733           HIFATOM EC001  ④
09AC              01737 EC005   EQU   $
09AC    2A923F    01738          LHLD  ZZARG1+2  ⑤
09AF    E5        01739          PUSH  H
09B0    2A903F    01740          LHLD  ZZARG1    ⑥
09B3              01741          GBULK1 90H
09B8    2A8D3F    01745          LHLD  BULKBUF+2 ⑦
09BB    E5        01746          PUSH  H
09BC    2A8B3F    01747          LHLD  BULKBUF
09BF    22DB3F    01748           SHLD  ZZADR1   ⑧
09C2              01749          GBULK1 91H     ⑨
09C7              01753          EVALE          ⑩
09C8    E1        01755           POP   H       ⑪
09C9              01756          DIFNIL EC002   ⑫
09D5              01763          GBULK1 90H     ⑬
09DA    2A8B3F    01767          LHLD  BULKBUF  ⑭
09DD    22DB3F    01768          SHLD  ZZADR1   ⑮
09E0              01769          GBULK1 91H     ⑯
09E5              01773          EVALE          ⑰
09E6              01775          TVKILL         ⑱
09ED    E1        01779          POP   H   ; DUMM  ⑲
09EE    C9        01780          RET            ⑳
09EF              01781 EC001   EQU   $
09EF    EB        01782          XCHG           ㉑
                  01783 ;      CALL ERRORA3
09F0              01784            TVKILL   ㉒
09F7    C9        01788          RET
09F8              01789 EC002   EQU   $
09F8    E1        01790          POP   H        ㉓
09F9              01791           HIFATOM EC001  ㉔
09FF              01795          GBULK1 91H     ㉕
0A04    C3AC09    01799          JMP   EC005
                  01800 ;
                  01801 ;
                  01802 ;    END OF INTERPRETER NUCLEUS
                  01803 ;
                  01804 ;
                  01805 ;-------------------------------
```

図 2 インタプリタ関数の一部である EVCON のソースリスト

(COND は if p_1 then e_1 else if p_2 then e_2 … を表わす)

① (COND (p_1* e_1*) … (p_n* e_n*)) の処理
② ZZADR1 には ((p_1* e_1*) … (p_n* e_n*)) のリストの先頭番地が与えられている
③ G.C. 保護用スタッキングをするマクロ
④ (H, L) レジスタの内容がアトミックならば EC001 へ Jump するマクロ
⑤ ((p_{i+1}* e_{i+1}*) … (p_n* e_n*)) を save
⑥ (p_i* e_i*) を0番バッファ (BULKBUF) へ読み出す
⑦ e_i* のリスト (CDR 部) を save
⑧ p_i* のアドレスを ZZADR1 にセット
⑨ 1番バッファ (ZZARG1) へ p_i* の内容の読出し
⑩ p_i* の評価
⑪ e_i* のリストを pop
⑫ EVAL した値 (p_i* の値；(D, E) レジスタに入れられている) が NIL なら EC002 へ
⑬ そのときの H-L で示される番地 (e_i* のリスト) の0番バッファへの読出し
⑭ CAR 部 (e_i*) の取出し
⑮ e_i* のアドレスを規約どおりにセット
⑯ e_i* の先頭番地の内容を1番バッファに読出し
⑰ e_i* の評価
⑱ TVSAVE されたものを KILL する
⑲ スタック上に残された ((p_{i+1}* e_{i+1}*) … (p_n* e_n*)) を捨てる
⑳ 値は EVAL した値が，(D, E) レジスタに残されているので，それを値としてそのまま帰る
㉑ p_n まで評価された (または空リスト引数が渡された) ので最後のセルの CDR 部 (おそらくは NIL) を値とすべく (D, E) レジスタへもっていく
㉒ TVSAVE されたものを KILL する
㉓ p_i* が NIL だったので $i+1$ を見にいく
㉔ もしそれがアトムなら ((p e) ペアがない) EC001 へいく
㉕ $i+1$ 部を1番バッファへ読み出す

* ヘキサデシマル・フォーマットは，インテル社で定めたフォーマットで，次の例のようなものである．

:1E400000C3CF12C33113C34A13C38B44C3
:1E401E00C30D46C32E46C37746C39F47C3
:1E403C00C38A0F02144F042A2A474F0100

① 先頭にコロン (:) コード
② 次の2桁でロードするバイト数
③ 4桁でロードの先頭番地
④ ゼロ2桁 (インテルでは for future use といっている)
⑤ ロードされるべきオブジェクト
⑥ チェックサム2桁

この①から⑥までが1単位となる．⑥の後に次の①がくるまでのあいだは何が入っていてもよい．また長さも任意である．終了レコードは②がゼロ4桁のものである．このとき③がゼロ4桁ならばモニタに返り，それ以外ならその番地へジャンプする．

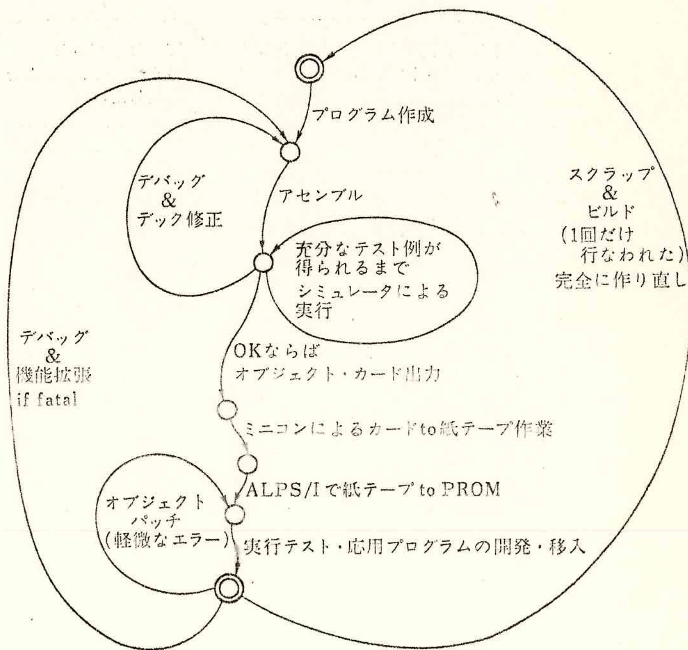

図 3 プログラム開発の流れ

表 1 シミュレータのコマンド

コマンド名	機　　能
$LOAD	/* または $END がくるまで，ヘキサデシマル・フォーマットのオブジェクトをシミュレータ上のインテル用メモリへロードする
$START および $STARTFAST	プログラムカウンタを0にして実行を開始する．シミュレート時にはバルクの Read/Write の回数，実行命令数，経過時間のカウント，命令ごとの使用頻度 (FAST 指定時はとられない)，命令の意味的なチェック (FAST 指定時はされない) が行なわれる
$BREAKPOINT	$BREAKPOINT {, アドレス [, {DUMPINTEL / DUMPBULK / DUMP / MONITOR}] / CLEAR} 指定されたアドレスに来ると，インテルメモリのダンプ (DUMPINTEL)，バルクメモリのダンプ (DUMPBULK)，その両方 (DUMP) およびモニタに返り，次のコマンドをもらう (MONITOR) ことを要求する．あわせてそのときまでのバルク Read/Write 回数，実行命令数，経過時間のカウント，レジスタの内容などが印刷される．ブレークポイントの解除には CLEAR 指定をする
$RETURN	モニタを呼んだルーチンへ戻る
$CONTINUE	実行ルーチンを呼ぶ
$DUMP {INTEL / BULK / JOB}	メモリのダンプを行なう INTEL …シミュレートされたインテルメモリのダンプ BULK …シミュレートされたバルクメモリのダンプ JOB …シミュレータ自身のダンプ
$TRACEBACK	現在実行した命令よりさかのぼって16命令を印刷する (なお，エラー時には自動的にトレースバックプリントがされる)
$TRACE および $TRACECLEAR	命令実行のトレースの開始および解除指定
$INSTRUCTION COUNTCLEAR	命令使用頻度カウンタを0にセット
$HALT	シミュレータ実行の停止

図2に示す．これは，Lisp 処理系の一部の EVCON と呼ばれる関数である．

プログラム作成の流れを図3に示す．アセンブルされたオブジェクトはシミュレータに直接渡すか，あるいは出力フォーマッタ・プログラムを通してヘキサデシマル・フォーマット*(前ページ脚注) のカードを出力する．出力されたオブジェクト・カードはミニコンで紙テープに変換される．この紙テープを ALPS/I に入力させ，PROM プログラマ・ルーチンを通して PROM に書き込み，1回分が完成する．

```
$LOAD
$BREAKPOINT,X'064C'
$STARTFAST
ALPS/I READY

EVALQUOTE ENTERED, ARGUMENTS...
  DEFINE((
   (EQUAL (LAMBDA (X Y)
     (COND((ATOM X) (EQ X Y))
          ((ATOM Y) NIL)

        (途中省略)

          (SETQ W (APPEND W (LIST V)))
          (GO A)
   )))
 ))
BREAKPOINT BR=   000688 BW=   009990 IC=     202951
                 329596 064C 210000 11 01000 00 04 20
END OF EVALQUOTE, VALUE IS..
(EQUAL APPEND SMALLER MINM DIFFLIST SEQUENCE)

EVALQUOTE ENTERED, ARGUMENTS...

SEQUENCE((11 17 3 16 17 3 19 32 22 39 31 33 46 29 9 40
BREAKPOINT BR=   000765 RW=   909344 IC=     213974
                 334050 064C 210000 18 01000 3F 04 20
END OF EVALQUOTE, VALUE IS..
(3 7 9 11 16 17 19 21 22 29 31 32 33 39 40 43 46)

EVALQUOTE ENTERED, ARGUMENTS...
 STOP
  MAX CELLS=35E4


   BYE
$HALT


*************************************************
*****  IBM SYSTEM/370 MODEL/138 DOS/VS  *****

           ACCOUNTING INFORMATIONS
     1.USER-NAME ........ TOHMINE TAKAYOSHI
     2.TIME-SET ......... 12   MINUTES
     3.PAGE-SET ......... 60   PAGES
     4.RUN-DATE ......... 09/11/78
     5.PARTITION ........ BG
     6.START-TIME ....... 10H 40M 05S
     7.STOP-TIME ........ 10H 48M 50S
     8.USED-STORAGE .....    151551 BYTES
     9.COMPILE-TIME .....      0.00 SEC (CPU)
    10.EXECUTE-TIME .....    207.33 SEC (CPU)
    11.OVERHEAD-TIME ....      9.49 SEC (CPU)
    12.ALLBOUND-TIME ....      0.11 SEC (CPU)
    13.PRINTED-SHEET ....         1 PAGES
    14.PUNCHED-CARDS ....         0 CARDS
    15.EOJ-STATUS-CODE ......... 10

********  AOYAMA GAKUIN UNIVERSITY  *********
*************************************************
```

図 4 シミュレータによる実行例

プログラムのデバッグ作業は，このようなことからも，ハード作成の片手間に行なうことはできそうもなかったので，75年11月はハード作成は中止し，学校に8泊ほどして集中的に行なった．シミュレータにはスナップショット・ダンプ機能などが入れられている．シミュレータに与えられるコマンドの一覧を表1に示す．実行例を図4に示す．思いつく限りの入力を行ない，それらの結果をまとめてデバッグする．同じ原因から生じる違った症状などから吟味しながらデバッグできたことになり，非常に効率はよかった．76年1月に初めて約3時間をかけて PROM への書込みを行なった．このとき，それらは

即座に動作し，いくつかのよく用いられる例題はすぐに実行できた．机上のデバッグとクロスソフトウェアの効果の大きさを感じる．このクロスソフトウェアは非常に使いよかったが，いま考えてみれば次のような点も指摘できる．

❶アセンブラは絶対形式でなく，相対アセンブルを行なえるほうがよい．ALPS/I のプロセッサはそれほど混み入ってはいない．最大 2K バイト以下のモジュールの集まりになっている．それらを独立にアセンブルできるほうが処理効率がよい．

❷リンケージ・エディタと相対番地プログラム・ライブラリ機能の付加．❶をとるならば必然的に必要．

❸ソースプログラム・ライブラリとエディタ．ソースプログラムをディスク上に持つことは諸々の制約でわれわれの場合はできなかった．アセンブルのたびにカードを流していたが，IBM のカードリーダは大変具合がよく，ほとんどカードをいためたり，エラーを起こしたりしないので何とかもっている．しかし，信頼性の低いカードリーダを持つほうが幸福かもしれない(!?)．ソースプログラム・ライブラリとエディタを作る必要性をもっと切実に感じられたかもしれないからだ．（しかし，その反動でパーソナルシステム志向が起きた？）

❹　クロスシステム上のカッコカウントルーチン．これは Lisp 用のものである．たとえば，((A B)) というリストを入力すると，

((A　B))
01　　10

のような印刷をするプログラムである．単独のプログラムとして作成したものを使っているが，クロスシステムと連動できるとさらに便利である．最初に Reduce を東大からいただいたときに，このプログラムにかけたところ，カッコの対応しない部分が発見され，これにより2枚のカードの誤順序が発見できた．この時点で検出されていなければ Reduce は動かなかったかもしれない．また，このとき pretty print できるほうが望ましい．

なお，リテラル（文字定数）中のカッコをカウントしないように注意しなければならない．Reduce をいただいた当時の HLisp のカッコカウントルーチンではリテラル中のカッコもカウントされていて，こちらの出力とつき合わせるとカウントがあわず，一瞬慌てた記憶がある．

❺Lisp ソースプログラム・ライブラリとエディタ．❸のものと同じでもよい．

❻高級言語．なんといっても高級言語（たとえば，C，PL/M，Lisp コンパイラ版）が欲しい．次期開発の際には高級言語で行ないたい．もっともこの ALPS/I のコーディングにはあまり不自由は感じなかった．なぜかというと，コーディングに先立ってM式でモジュールを定義し，それをコメントとして書き込む．あとはそれを見ながら機械的に作成（ハンドコンパイル）していったら約8千枚になってしまったからである．

3.　デバッグはお茶を飲んでから

標題は若干誤解されるかもしれないが，言いたいことは，改良・修正は小刻みには行なわずに，ゆっくりとお茶でも飲んで全体をながめ，どこを直すのが本質的か，方針変更の要はないかの吟味からはじめることが必要だということである．まったく，あわてて直したものに良いものはあまりない．多くの場合，全部つくりなおしてもかまわないという気でデバッグするとプログラムも見やすくなり，かつ高速になる．

76年1月に書き込んだ初版は半年の命だった．いくつかの改良点が初版で指摘されたが，それらを段階的にほどこす気になれなかったので，初版の約5千枚はばっさり捨て去り，気をとりなおして最初からコーディングをしなおした．やりなおす気になると不思議なもので欲がでて，RST 命令の利用によるサブルーチン・コールの高速化，アトムや数の高速判別など，入れたいと思っていたものがだいたい入ってしまった．また，アトムの car や cdr をとることをソフトウェア的にガードしたり，デバッグ用の処理系トレース機能をインタプリタ・ループ中に組み込んだりした．このため，インタプリタ本体は若干重くなり，オーバーヘッドが増すと思われたが，アルゴリズムの高速化の効果のほうが勝ったようで，逆に1.5倍程度，初版より速度が向上した．もっともデバッグのたびに8千ステップのソース中にどこか改良できる所はないかと「いきがけの駄賃」を求めた効果が多少なりとも影響していると思われる．（8080 のプログラミング・テクニックはいくつかの本にも紹介されている．あまりこったもの，たとえば「命令の中の命令」などは好ましくないが，命令セットの特長を利用することは当然必要である．それらについては，たとえば，**bit**の増刊号「マイクロコンピュータのプログラミング」に中西正和先生が書かれている．）

4.　マイコンの命令は Lisp 向き？

8080 命令セットは他の言語をのせる場合に比べて，Lisp の場合には比較的向いているのではないかという


```
00001 HIFNIL   MACRO HN         ┐
00002            MVI   A,14H    │
00003            CMP   H        │
00004            JNZ   $+9      │ ①
00005            MVI   A,92H    │
00006            CMP   L        │
00007            JZ    HN       │
00008            ENDM           ┘
00009 DIFNIL   MACRO DN         ┐
00010            MVI   A,14H    │
00011            CMP   D        │
00012            JNZ   $+9      │ ②
00013            MVI   A,92H    │
00014            CMP   E        │
00015            JZ    DN       │
00016            ENDM           ┘
00017 EVALE    MACRO            ┐
00018            RST      7     │ ③
00019            ENDM           ┘
00020 HIFATOM  MACRO HA         ┐
00021            MVI   A,1FH    │
00022            CMP   H        │ ④
00023            JNC   HA       │
00024            ENDM           ┘
00025 HIFSML   MACRO HS         ┐
00026            MVI   A,3      │
00027            CMP   H        │ ⑤
00028            JNC   HS       │
00029            ENDM           ┘
00030 GPULK1   MACRO COMAND     ┐
00031            DW    2670H    │
00032            DB    COMAND   │ ⑥
00033            DW    6777H    │
00034            ENDM           ┘
00035 GPULK2   MACRO            ┐
00036            DB    7CH,60H,77H,67H  │ ⑦
00037            ENDM           ┘
00038 OBLIS    MACRO NAME,CNT,CC,ATRIB  ┐
00039            DB    ATRIB    │
00040            DW    CC       │ ⑧
00041            DB    CNT,NAME │
00042            ENDM           ┘
00043 ADVACE   MACRO            ┐
00044            CALL GETTTY    │ ⑨
00045            ENDM           ┘
00046 PUTTTY   MACRO            ┐
00047            RST   1        │ ⑩
00048            ENDM           ┘
00049 HIFZ     MACRO    HZ      ┐
00050            MOV   A,H      │
00051            ORA   L        │ ⑪
00052            JZ    HZ       │
00053            ENDM           ┘
00054 TVSAVE   MACRO            ┐
00055            CALL TVSAVE00  │ ⑫
00056            ENDM           ┘
00057 TVRESTOR MACRO            ┐
00058            RST   5        │ ⑬
00059            ENDM           ┘
00060 TVKILL   MACRO            ┐
00061            LHLD     TVPOINT  │
00062            DCX   H        │ ⑭
00063            SHLD     TVPOINT  │
00064            ENDM           ┘
00065 TVKILL2  MACRO            ┐
00066            LHLD     TVPOINT  │
00067            DCX   H        │
00068            DCX   H        │ ⑮
00069            SHLD  TVPOINT  │
00070            ENDM           ┘
00071 ERRSETP  MACRO            ┐
00072            LDA   ERRSW    │
00073            RAL            │
00074            JNC   $+7      │ ⑯
00075            LXI   D,NIL    │
00076            RET            │
00077            ENDM           ┘
00078 PHASHINT MACRO            ┐
00079            LXI   H,0      │ ⑰
00080            SHLD  INCRHASH │
00081            ENDM           ┘
```

① (H, L) レジスタの内容が NIL か？
NIL はシステム定数で，16進では1492である（現在は）．
この値はハッシュ関数のアルゴリズムにより定められる．
NIL ならばオペランドに書かれたところ，HN に飛ぶ

② (D, E) レジスタの内容が NIL か？
NIL ならばオペランド DN に記されたところに飛ぶ

③ 現在の第1引数バッファ上のものに対してインタプリタの eval を働かせよ．
eval は16進38番地から入れられているので RST 命令によるサブルーチンコールを使用できる

④ (H, L) レジスタ中のポインタ（バルクアドレス）はアトミックか否か．
アトミックならば HA へ飛ぶ．
（アトミックな情報は16進2000番地より前に入れられている．リスト領域は2000～FFFF 番地）

⑤ (H, L) レジスタ中のポインタは基本整数か？
（基本整数の場合（0～1023），ポインタ自身が値となる）

⑥ バルク Read/Write コマンド1
（(H, L) レジスタが示すバルクアドレスに対して COMAND で指定する転送を行なう）
引数 COMAND の上位 1ビット＝1
　　3ビット＝インターフェイスコマンド
　　4ビット＝AAM ナンバー＝バルクバッファナンバー

⑦ バルク Read/Write コマンド2
（(H, L) レジスタが示すバルクアドレスに対して，レジスタ B にあるコマンドにより転送を行なう）

⑧ システム組込みアトムの定義用
（作られた定義は初期化ルーチンによりバルク上のハッシュ領域にセットされる）
例　OBLIS 'ATOM', 4, ATOM, 1
意味　4文字 'ATOM' を pname とするアトムを属性が1（SUBR），値として ATOM（処理ルーチン先頭番地）を入れて定義する．

⑨ 1字入力装置からもらう．入力された文字の ASCII コードがレジスタ C に返される

⑩ レジスタ C の内容をそのまま印字する

⑪ (H, L) レジスタの内容がゼロか否か
ゼロなら HZ（オペランド）へ分岐

⑫ (H, L) レジスタの内容をスタック
(H, L) は Lisp データでありガーベジ・コレクタ保護の必要があるので push 命令を使わない．ソフト的にバルクメモリへスタックする

⑬ TVSAVE で push されたスタックから (H, L) レジスタへ pop up

⑭ TVSAVE のスタックの最上部の情報の除去
（ポインタを1へらすだけ）

⑮ TVSAVE のスタックの最上部の2つを除去

⑯ errorset〔..., T〕のためのエラーメッセージのバイパス処理の埋込み

⑰ Rehash カウンタの初期化

図5　ALPS/I で使用したマクロ定義

気がする．

まず第一に，再帰的な手続きを記述するのが容易である．すなわち，call，push，pop，xthl* などの命令が備わっており，これらを利用してコンパクトにまとめられる．第二に，Lisp の場合には本来単純な操作の積重ねが多い．特徴である二進木リストはポインタの対であり，処理系が扱う情報の多くはポインタとして使われている固定長のデータである．ALPS/I の場合，Lisp データは 64K 語のバルクメモリにすべて入れられており，そのそれぞれはバルクアドレス（16ビット）で示される．そしてこの16ビットの情報を比較したり転送したりする仕事が多い．（逆にいえば，16ビットで Lisp データを

*　たとえば，push は 7 μ 秒，pop は 6 μ 秒，xchg は 2.5 μ 秒，call は 10 μ 秒を要する．

```
                      01907 ;
                      01908 ;   **********  ATOM PREDICATE **********
                      01909 ;               IF ARGS IS LESS THAN 2000 THEN TRUE
09F5                  01910 ATOM     EQU    $
09F5      3ADC3F      01911           LDA    ZZADR1+1
09F8      FE20        01912           CPI    20H
09FA      DA010A      01913           JC     SETTRUE
09FD      119214      01914 SETNIL   LXI    D,NIL
0A00      C9          01915           RET
0A01      110019      01916 SETTRUE  LXI    D,TRUE
0A04      C9          01917           RET
```

図 6 関数 atom

```
                      02075 ;
                      02076 ;   **********   CAR------   CDDDR **********
                      02077 ;
0AFF                  02078 CAR     EQU    $
0AFF      2ADB3F      02079           LHLD   ZZADR1        ①
0B02                  02080 CAR00    EQU    $
0B02                  02081           HIFATOM ERRORATM  ┐
0B08      2A903F      02085           LHLD   ZZARG1     │②
0B0B      EB          02086           XCHG              ┘
0B0C      C9          02087           RET
0B0D                  02088 CDR     EQU    $
0B0D      2AD93F      02089           LHLD   ZZADR1
0B10                  02090 CDR00    EQU    $
0B10                  02091           HIFATOM ERRORATM
0B16      2A923F      02095           LHLD   ZZARG1+2
0B19      EB          02096           XCHG
0B1A      C9          02097           RET
0B1B                  02098 CAAR     EQU    $
0B1B      2ADB3F      02099           LHLD   ZZADR1        ③
0B1E                  02100 CAAR00   EQU    $
0B1E                  02101           HIFATOM ERRORATM     ④
0B24      2A903F      02105           LHLD   ZZARG1        ⑤
0B27                  02106           GBULK1 91H         ⑥
0B2C      C30208      02110           JMP    CAR00      ⑦
0B2F                  02111 CADR     EQU    $
0B2F      2ADB3F      02112           LHLD   ZZADR1
0B32                  02113 CADR00   EQU    $
0B32                  02114           HIFATOM ERRORATM
0B38      2A923F      02118           LHLD   ZZARG1+2
0B3B                  02119           GBULK1 91H
0B40      C30208      02123           JMP    CAR00
```

① 引数を (H, L) レジスタへ
② H-L は素か？
素でないので，car 部を取り出し，値とする
③ 引数を (H, L) レジスタへ
④ H-L は素か？
⑤ car 部を取り出す
⑥ そのアドレスのバルクメモリを第1バッファ (ZZARG1) に読み込む
⑦ さらに car をとる

図 7 関数 car, cdr, caar, cadr

識別し，処理できるようなインターフェイスを作ったので，バイトアドレシングしかできないマイクロコンピュータでも語処理が可能となった.）

例をとおしてこれらのことを見てみよう．

図5に ALPS/I で使用しているマクロを示す．全部で17個ある．それぞれは MACRO 文に始まり，展開される原形文そして ENDM 文で1区切りとなっている．MACRO 文の第1フィールドには定義されるマクロ名を記述し，オペランド・フィールドには，もしあるのなら，マクロ命令呼出し時の引数を記述する．

たとえば，図2のソースリストの3行目「HIFATOM EC001」は，

「MVI A,1FH」「CMP H」「JNC EC001」

の3命令に落とされる (ALPS/I での実行時間は 13.5 μ 秒).

また Lisp データの転送のために2つのマクロ GBULK1 および GBULK2 を用意している．

図2の下から2行目の「GBULK1 91H」は，(H, L) レジスタで示されるバルクアドレスから，RAM の1番バッファ（5バイト）へ読み込むことを表わしている．バッファは16通りとることができ，その番号をオペランドの下1桁で示す．実際のバッファアドレスは前もって，GBULK コマンドを使って登録しておく．上1桁は現在，'9' か 'A' か 'C' しかない．9ならば Read，AならばWrite，C ならばバッファアドレスのインターフェイス内の AAM レジスタへのセットを表わしている．バルクアドレスはそのときの (H, L) レジスタにより与えられる．このようにすることによって，頻繁に加工を受けるバルクアドレス情報を push，pop その他の命令を効果的に組み合わせて処理することができる．

次に，Lisp で用いられる組込み関数 (car，cdr など) のコーディングの際に用いた規約を説明しよう．次のようなものである．

❶引数は，呼び出す手続きにおいてバルクバッファ (ZZARG1, …, ZZARG14) 上にすでに並べられているものとする．たとえば，2引数ならば第1および第2バッファに入れられる．さらにそのバルクアドレスは RAM 上の対応するアドレスレジスタ (ZZADR1, …, ZZADR14) に入れられているものとする．

❷関数の出口において，値は (D, E) レジスタに入れて帰る．

❸すべての 8080 レジスタの save，restore は，原則として行なわない．

❹シュードコーディングとしてM式で動作を記述し，それに対応させてコーディングを行なう．M式はコメントとしてソースプログラム中に必ずいれる．

❺関数内で必要となる作業域は，入出力バッファ，他とも共通して用いられるフラグ類を除いて，とってはならない（スタックを使うこと)．

このような規約のもとにコーディングのいくかを見てみよう．(なお Lisp に関する知識は，**bit** 誌連載の「Lisp 手習」を参照してほしい.）

Lisp の基本関数の中で atom というものがある．これは，与えられた引数が素であるか，リストであるかを判別し，素であれば真 (T) を，そうでなければ偽 (NIL) を返すものである．ALPS/I では Lisp データは，すべてバルクメモリ上にあり，そのアドレスによって扱われることは前に述べた．バルク上のリスト領域（フリーストレージ）は 2000_{16} 番地以降にとられており，素な情報は $1FFF_{16}$ 番地より前に入れられている．このため関数 atom は，与えられた引数のアドレス値が 2000_{16} より前ならば真になるようにすればよい．そのプログラムは図6のようになる．

ZZADR1（2バイト）に第1引数のバルクアドレスが入っている．上位1バイトをAレジスタにロードする．8080の仕様から16ビットデータは上下さかさまになっているので，「LDA ZZADR1+1」がこれにあたる．(なお，ZZ という接頭語は見やすさのためにバルクバッファを表わす ZZARG n (各5バイト) および ZZADR n (各2バイト，$1 \leq n \leq 14$) に対してつけられている.）CPI 命令で判別したのちに，関数値を返す (D, E) レジスタに T または NIL を入れる．これらの部分は，他からも JMP され，共有されるので，おのおの SETNIL，SETTRUE のラベルがつけられている．

図7に car～cadr のコーディングを示す．素な引数の car および cdr をとろうとするとエラーがでるようになっている．実行時間は若干損にはなるが，実用上この程度のことは入れておかないと，デバッグなどに支障をきたす．

リストを作成するための基本的な関数としてconsがある．cons は2引数を必要として，第1引数を car 部に，第2引数を cdr 部にもつドット対をバルクに書き込み，そのアドレスを値として返すように作成する．図8にそのコーディングを示す．このコーディングで使用している「GETCELL」というサブルーチンはフリーストレージから未使用のセルを1つとってきて，そのアドレスを (H, L) レジスタに返すものである．未使用セルがなくなったら，このルーチン中でガーベジ・コレクタが呼び出される．

図9にマッカーシーが定義した基本5関数の最後の1つである eq を示す．eq は本来素な引数に対してだけ定義されているが，実際的には多くの処理系で，すべての Lisp データに関する同値 (equivalent) をチェックするように拡張されている．ALPS/I でもすべての Lisp データに対して eq をかけることができる．その仕事は与えられた2引数のアドレスを比較し，等しければTを返し，そうでなければ NIL を返すことである．

すべての関数は上に述べた atom，car，cdr，cons，


```
                      02334 ;
                      02335 ;   **********   CONS   **********
                      02336 ;
0C89                  02337 CONS    EQU    $
0C89      2ADB3F      02338           LHLD   ZZADR1        }①
0C8C      22D63F      02339           SHLD   BULKBUF2      }
0C8F      2ADD3F      02340           LHLD   ZZADR2        }②
0C92      22D83F      02341           SHLD   BULKBUF2+2    }
0C95                  02342 CONS1   EQU    $
0C95      AF          02343           XRA    A             }③
0C96      32DA3F      02344           STA    BULKBUF2+4    }
0C99      CDF747      02345           CALL GETCELL       ④
0C9C      54          02346           MOV    D,H         }⑤
0C9D      5D          02347           MOV    E,L         }
0C9F                  02348           GBULK1 AFH         ⑥
0CA3      C9          02352           RET
```

① 第1引数を car 部に
② 第2引数を cdr 部に
③ フラグ部のクリヤー
④ 一つセルをもらってくる
⑤ もらったアドレスを値用にセット
⑥ そのアドレスに15番バッファ (BULKBUF2) から write する

図 8 関数 cons

```
                      02748 ;
                      02749 ;   *********   EQ **********  PREDICATE   SUBR 2 ARGS
                      02750 ;
                      02751 ;     ATOM,NUMBER,HARRAY,ASSOCCOMP ARE EQ-ABLE
                      02752 ;
                      02753 ;
0ECC                  02754 EQ      EQU    $
0ECC      2ADB3F      02755           LHLD   ZZADR1
0ECF      3ADE3F      02756           LDA    ZZADR2+1
0ED2      BC          02757           CMP    H
0ED3      C2FD09      02758           JNZ    SETNIL
0ED6      3ADD3F      02759           LDA    ZZADR2
0ED9      BD          02760           CMP    L
0EDA      C2FD09      02761           JNZ    SETNIL
0EDD      C3010A      02762           JMP    SETTRUE
```

図 9 関数 eq

```
                    02763 ;
                    02764 ;  **********  FQUAL  **********  PREDICATE SUBR
                    02765 ;
                    02766 ;   EQUAL(X,Y)=(ATOM(X)->EQ(X,Y); ATOM(Y)->NIL;
                    02767 ;          EQUAL(CAR(X);CAR(Y))->EQUAL(CDR(X);CDR(Y));
                    02768 ;          T->NIL;
0EE0                02769 EQUAL     EQU    $
0EE0    2ADB3F      02770           LHLD   ZZADR1     } ①
0EE3                02771           HIFATOM EQ
0EE9    2ADD3F      02775           LHLD   ZZADR2     } ②
0EEC                02776           HIFATOM SETNIL
0EF2                02780 EQUAL01    EQU    $
0EF2    2A923F      02781           LHLD   ZZARG1+2   } ③
0EF5    E5          02782           PUSH   H
0EF6    2A973F      02783           LHLD   ZZARG2+2   } ④
0EF9    E5          02784           PUSH   H
0EFA    2A903F      02785           LHLD   ZZARG1     }
0EFD    22DB3F      02786           SHLD   ZZADR1     } ⑤
0F00                02787           GBULK1 91H        }
0F05    2A953F      02791           LHLD   ZZARG2     }
0F08    22DD3F      02792           SHLD   ZZADR2     } ⑥
0F0B                02793           GBULK1 92H        }
0F10    CDE00E      02797           CALL   EQUAL        ⑦
0F13                02798           DIFNIL EQUAL03
0F1F    E1          02805           POP    H          }
0F20    22DD3F      02806           SHLD   ZZADR2     } ⑧
0F23                02807           GBULK1 92H        }
0F28    E1          02811           POP    H          }
0F29    22DB3F      02812           SHLD   ZZADR1     } ⑨
0F2C                02813           GBULK1 91H        }
0F31    CDE00E      02817           CALL   EQUAL      ⑩
0F34    C9          02818           RET
                    02819 ;
0F35                02820 EQUAL03    EQU    $          ⑪
0F35    E1          02821           POP    H ;    DUMMY
0F36    E1          02822           POP    H    ;    DUMMY
0F37    C9          02823           RET
```

① atom[x]→eq[x; y]
② atom[y]→NIL
③ cdr[x] の save
④ cdr[y] の save
⑤ car[x] を第1引数としてセット
⑥ car[y] を第2引数としてセット
⑦ equal[car[x]; car[y]] ならば⑧, ⑨へ. そうでなければ⑪へ
⑧ cdr[y] を第2引数にセット
⑨ cdr[x] を第1引数のセット
⑩ equal[cdr[x]; cdr[y]] をして, そのままの値で帰る (JMP EQUAL でも同じ)
⑪ T→NIL

図 10　関数 equal

eq の合成によって記述できるというのが最初の文献* に述べられており, ここにも Lisp は数学的な美しさがあるといわれる一因がある. しかし実用的にはさらにいくつかの関数をシステムに組み込んでおかないと, 使用者にとっては不便であり, また速度も遅くなる.

図10に関数 equal のコーディングを示す. ALPS/I ではこのような形で (頭に注釈として名称, M式での定義をつけ) 約100の関数を組み込んでいる. コーディングはM式での記述を直接ハンドコンパイルするつもりで作成されている.

関数 equal は2つの引数を必要とする. 与えられた2つのリストの全枝葉が等しい場合に equal は真 (T) を, そうでなければ偽 (NIL) を返す.

equal は基本的には次のように定義される.

equal[x; y]=[atom[x]→[atom[y]→eq[x; y]
　　　　　　　　　　　T→NIL];
　　　　atom[y]→NIL;
　　　　equal[car[x]; car[y]]→equal
　　　　　[cdr[x]; cdr[y]];
　　　　T→NIL]

(x が素ならば, もし y が素であれば eq[x; y], そうでなければ NIL. y が素ならば NIL (∵ x が素でなく y が素ならば等しいことはありえないから). car どうしが等しければ cdr どうしを比べよ. car どうしが等しくなければ NIL. 等しければ cdr どうしを比べよ.)

ここで下線部は, eq という関数は素な引数に対してのみ定義されていることから生じたものである. しかし前述したように多くの処理系では (そして ALPS/I でも) すべての Lisp データを eq で調くことを許しているので, その部分をただ eq[x; y] で置き換えることができる. 図10中のM式はその段階での定義となっている.

第1引数 x が素ならば EQ へ分岐する. x が素でなく, y が素ならば偽であるから SETNIL へ飛ぶ. x と y はそれ以外の場合リストであるので, 再帰的呼出しを行なって car 部, cdr 部をそれぞれどちらかが素になるまで繰り返す. 通常の再帰的呼出しのための手順では引数をスタックして行なうが, ALPS/I の場合では引数のアドレスの内容は読み出された状態で各ルーチンに入ってくる約束になっている. このため, 引数の car および cdr は, バルク参照をかけることなしに LHLD 命令だけで取り出すことができる. equal 関数の場合, save が必要なのは car 部へ再帰的呼出しをしているときの, 対応する cdr 部の値であるので, それをそのまま push 命令でスタックする. そして x, y の car 部を引数としてセットし, equal を再帰的呼出しする. car 部が等しければ, cdr 部を見にいく必要があるので, save しておいたものを pop し cdr 部の equal を見る. (なお, この説明を書くうちに無駄なコーディングの部分が見つかった. call equal→ret という部分である. これは Jmp equal に置き換えることができ, かつ高速になるので早速修正を行なった.)

そのほか, map, maplist, mapcon などのルーチンを紹介しようと思ったが, 紙面に限りがあるので, 別の機会にゆずることにする. また, 即時回収コレクタ** も evlis の定義を若干変更することによって組み込みやすくなる, とかの Lisp 処理系寄りの話も別の機会にゆずることにする.

以上, この章では ALPS/I でのプログラムを見てきたが, 8080 と Lisp のそれぞれについての基礎知識については特に解説をしなかったので, それについては他の文献を見ていただきたい.

5. 最後に

ALPS/I システムはバルクメモリないしはディスクなどの二次記憶があれば, ソース中のバルクコマンドマクロの中味を入れ換えることによって他の構成にも使えるようになっている. インタプリタ本体は 3K バイト程度, 雑サブルーチンが 2K バイト程度, 残りは組込み関数のコーディングである. 必要があればソースは喜んで公開する. ご連絡ください.

話はかわるが, 図11は数年後にはおそらくできるであろうポータブル Lisp マシンの絵である. 微分や積分のできる電卓なんていうのはいかがであろうか?

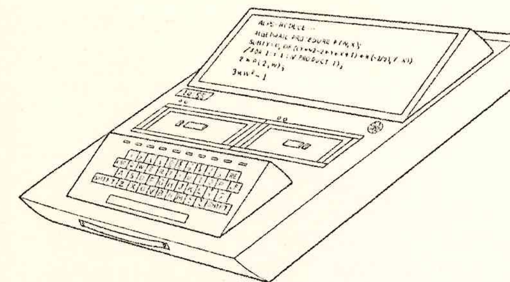

図 11　夢のポータブル Lisp マシン想像図

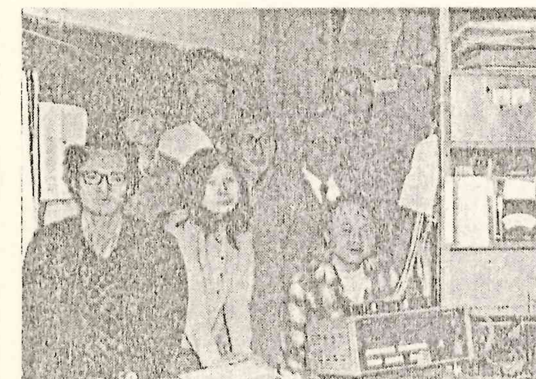

図 12　ALPS/I と格闘した面々 (前から三列目左: 筆者)

謝辞: 日ごろよりご指導いただく青山学院大学理工学部経営工学科間野浩太郎教授, 良きディスカッサントとしてそして良き助手として手伝ってくれた同研究室小林茂男・重光宏之・山方宏修・森芳喜・善本正一君に感謝する (図12).

記号処理研究委員会およびその後の研究会における討論は大いにはげましとなった. 〔完〕

(いだ まさゆき　青山学院大学)

* J. McCarthy: Recursive functions of Symbolic Expression and Their computation by machine.
** 後藤英一: 連載 Lisp 入門, **bit**, 1974参照.

㊄ 産業図書

パルス回路
猪瀬　博校閲 / 後藤公雄著 / 2100円
本書は、大学・高専の教科書、一般技術者等の参考書として、主に半導体素子を用いた最新かつ基本的パルス回路の動作を懇切丁寧に説明する。

数理解析とその周辺
編集委員 / 南雲仁一・藤田 宏・古屋 茂・山口昌哉・吉田耕作
23 工学における 非線形偏微分方程式 1上
W.F. エイムズ著/三村昌泰・小西芳雄訳/1900円
理・工学上の種々の非線形現象を記述している偏微分方程式を理解するために必須な解析的諸方法を詳しく解説した書。
24 工学における 非線形偏微分方程式 1下
W.F. エイムズ著/三村昌泰・小西芳雄訳/2600円
理・工学上に現われる非線形偏微分方程式の近似解法および数値解法を豊富な具体例を駆使して解説した実践書。

〈コンピューター サイエンス・ライブラリー〉
超大型コンピューター・システム
石田晴久・村田健郎著 / 2300円
コンピューター・アーキテクチャ
山田　博著 / 3000円
マイクロプログラミング
萩原　宏著 / 3300円

〈システム サイエンス シリーズ〉
編集委員 / 高橋秀俊・南雲仁一・伊理正夫
動的システムの理論　木村英紀著 / 2100円
適応システム　森下　巌著 / 1800円
しきい論理　室賀三郎・茨木俊秀・北橋忠宏著 / 2500円
カルマン・フィルター　有本　卓著 / 2300円
神経回路網の数理 ―脳の情報処理様式―
甘利俊一著 / 3000円
東京都千代田区外神田1―4―21
Tel 253-7821(代)/振替東京2-27724