

A L P S / II の LISP と アセンブラ

佐藤 衛・間野 浩太郎・井田 昌之

(青山学院大学 理工学部)

1. はじめに・概要

間野研究室では、インテル8080を用いたLispマシンA L P S / I を既に開発した [IDA79] が、現在その発展としてより大きく速いLispマシンA L P S / II を開発中である。A L P S / II の全体的概要については、[MAN82]で述べた。A L P S / II のLisp処理の中心となるL F U (Lisp Function Unit) のハードウェア上の特徴については、[SAT82a]、[SAT82b] で既に述べたが、簡単に紹介をする。L F U には3つのハードウェア上の特徴がある。それは

◎ ハードウェアスタックのPUSH DOWN と算術・論理演算の並列性

◎ マイクロ命令フェッチと実行の並列性 (条件ジャンプ時にも行なわれる)

◎ リスト (セル) アクセスと他の処理との並列性

である。最初の特徴は他の文献で触れていないので、この場で紹介する。フレーミング機能を持ったハードウェアスタックへの書き込みを算術・論理演算の結果の書き込みと並列化したのは、引数や変数、戻り番地のセーブの際は、ほとんどの場合、使用されていたレジスタを新しい値で書き換える処理が伴うからである。それに比べてセーブした値を元のレジスタにリストアするときはレジスタに入っていた値は捨ててよいはずなのでハードウェアスタックからの読み出しの場合には並列化は必要ない。スタック自身には、局所変数の取り扱いのためにスタックメモリに対し、ハードウェアで [ポインタレジスタの内容 + 即値のアドレッシング] を可能にしている。スタックにこの機能が付いたことで、スタックメモリのサイクルタイムとA L U のサイクルタイムがほぼ等しくなり、スタックとA L U の並列化によって時間を節約することができる。

また、A L P S / II のシステム全体についての特徴として、バックエンドプロセッサ (L F U) とフロントエンドプロセッサ (8086) の協調動作が挙げられる。L F U とフロントエンドの基本的な役割分担については [IDA82] に述べられている。この役割分担は、

◎ バックエンドプロセッサとフロントエンドプロセッサの並列動作

により、全体的処理効率の向上を目指すものである。その後、実際の処理系の作成に入ってから、メモリ効率、速度面から最適化のための改良が行なわれた。この問題については本発表の3. で詳しく述べられる。

2. A L P S / II - L I S P 処理系の概要

◎ マイクロプログラム化されたLISP

A L P S / II のLISP処理系は、基本的にはA L P S / I のLISP処理系の延長と言える処理系である。ただし、数式処理システムREDUCE-2のインプリメントを考えているのでStandard-lispの仕様を大幅に取り入れてある。合わせた仕様の部分は、大きなものとしては

◎ 変数束縛、(Global, Fluid, Local 変数のとりあつかい)

◎ トップレベル、(EVAL, ALPS/IはEVALQUOTE)

◎ 個々の関数の仕様

があげられる。また、仕様が合わない点は、

◎ 現在は、A L P S / II の『環境』はシステムの管理しているものだけである。EVALの

引数にa-listは書けない。

◎数値は今のところ小整数だけ。

◎メモリ割り当てなどの関数は勿論まったくない。

◎プロパティ関係の機能は完全にコンパティブルではない。

ことなどがあげられる。

A L P S / IIはまた、会話型であるから、プログラムのダイナミックデバッグができなければその魅力も半減する。そのためのツールとして、A L P S / Iでも一部採用されていた、

◎EVAL・APPLYの入口・出口で、その実引数をプリント出来るWALK機能、

◎関数を指定してTRACE(関数名・仮引数・実引数・値などをプリント)する機能に加え、A L P S / IIでは、

◎ダイナミックに(処理途中で)それらの機能を起動し停止させることを可能にする機能

◎A L P S / Iではプリントされなかったトレース時の実引数もプリントさせる機能

◎GLOBAL・FLUID変数、配列要素のトレースをする新機能

が備えられた。

これらの機能は、マイクロプログラムで制御するフラッグ(LFU内部の状態レジスタの中にあるフラッグ)でON・OFFされるので、ほとんど実行時間の負荷になっていない。

A L P S / IIのLISPの特徴の1つは、マイクロ命令レベルで再帰呼び出しが可能なので、処理系・基本関数が簡潔にマイクロプログラム化できるということである。ハードウェアとしては、マイクロプログラムの領域は充分大きく取ってある(最大32k語、インタプリタ・基本関数100個程をインプリメントして現在2k語程使用中)ので、将来、「LISP⇒マイクロプログラム」コンパイラを作成してコンパイル結果を走らせることも考慮中である。

◎ALPS/IIのデータ型

A L P S / IIがサポートするデータ型には、①リストを構成するCONSセル、②数値、③記号アトム、④配列要素の計4種がある。③の記号アトムには、①函数、②広域変数・定数(Standard LispのGlobal Variable等に対応)、③配列、④文字列、⑤入出力函数(フロントエンド手続呼び出し)が含まれているので、A L P S / IIの処理系は総計8種のデータ型を持っていることになる(図1)。

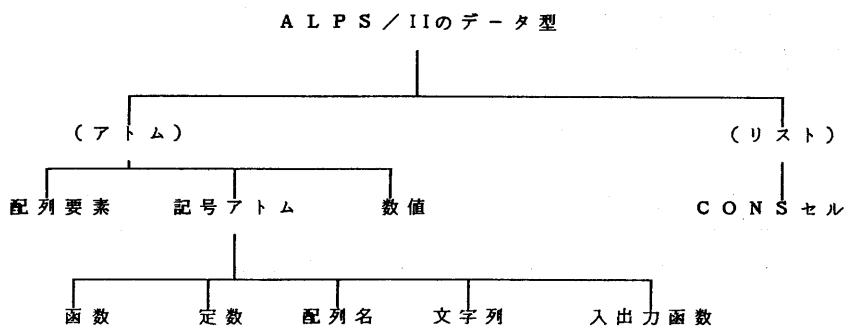


図1 A L P S / IIのデータ型

◎各データ型

現状では、取り扱えるCONSセルは最大0.5メガセルまでである。また、数値は[

- 2 6 2 1 4 4 , + 2 6 2 1 4 3] の整数にかざっている。この整数は記憶を費やさず、ポインタの値をそのまま用いている。現在は多倍長整数と浮動少数点数はインプリメントしていないが、追加する予定である。配列は、連続領域を必要とするいわゆる『配列』ではなく、ALPS / I でも用いられたハッシュ化配列 [IDA79] である。この配列機構によって、属性リストの取り扱いをする。記号アトムと配列要素とは同一の領域に格納される、現在扱えるのは両方合わせて 16 K 個までである。文字列も記号アトムの一つである。

◎ 入出力函数

入出力函数は、フロントエンドシステム内の手続に対応している、記号アトムの 1 つである。入出力函数の実体は、フロントエンドシステムに渡すメッセージである。たとえば、記号アトムをプリントする入出力函数を評価すると、インタプリタは、引数になっているアトムの P-name をフロントエンド側へ送り、次に print-atom というメッセージをフロントエンド側へ送る (図 2)。他の LISP 函数とおなじように、引数の個数の情報もアトムの中に持っている。

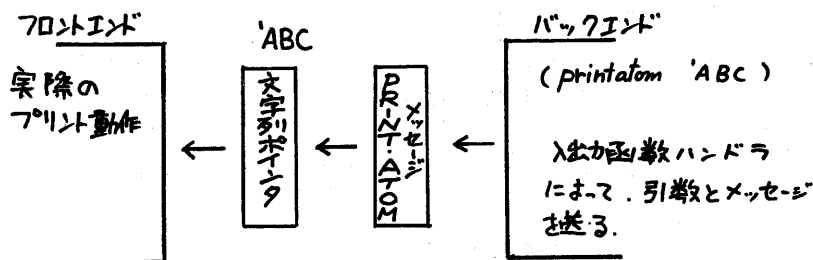


図 2 入出力函数によるメッセージ print-atom の送出

3. バックエンド (LFU) とフロントエンド (8086) との協調動作

実際の ALPS / II では、([IDA82] で示された) アトムはすべてフロントエンドシステム側に置くという方式とは異なったアトムの処理方式を用いている。すなわち、アトムの値や属性などはバックエンド (LFU) 側に置き、P-name・文字列だけをフロントエンド (8086) 側に置く方式である。これは、配列要素 (連想子として用いることがある) の取扱いや、アトムの値を引いて来る操作の効率を上げるためである。

◎ 入出力に伴う文字処理はフロントエンドに行なわせるのが得策

バックエンドシステム (LFU) は、高速なリスト処理用に設計したシステムである。文字列処理 (バイト単位の処理) などは、LFU にとって不得手な処理である。また、入出力に対する命令や機構も特に設けていない。これらはフロントエンド側で行なうので、文字列処理をバックエンド側で行なうことにすると、フロントエンドシステムとバックエンドシステムとの間の通信内容が複雑になる。そこで文字列そのものはフロントエンド側のメモリに格納する (例外がある、後述する)。バックエンド側へ引き渡し、バックエンド側のメモリ (バルクメモリ) に格納するのは、そのポインタだけとした。どちらのシステムも「得意」な分野の処理をすればよいことになる。すなわち、8086 はバイト単位の処理を、LFU はワード (20 ビット) 単位の処理を受け持っていて、全体として効率の良い処理が可能となる。

文字列だけでなく、ALPS / II の扱うデータがどのように格納されるかを図 3 に示す。

格納場所

データ型	Back-end	Front-end
	L F U	8 0 8 6
記号アトム (函数、定数、 配列名)	記述子 (属性、値、 文字数・ポインタ)	文字列
リスト	セル	-----
数値	整数	-----
配列要素	記述子 (配列名、添字、値)	-----
入出力函数	メッセージ (属性・ポインタ)	ファイル記述子・手続

図3 A L P S / IIの扱うデータ型による格納場所の違い

◎入出力の時だけでなく、他の処理にも応用できる

このように、フロントエンドシステムとバックエンドシステムにまたがってデータを格納することにより、入出力処理 (READ、PRINT 等) のみならず文字列にまつわる他の処理 (EXPLODE、COMPRESS等) もフロントエンドシステムとバックエンドシステムとの協調動作で行なえる。そのような処理の分割はどのようにして可能であるか。その基本機構の考え方は [IDA82] にすでに示されていることは前述した。 [IDA82]では、アトムと非アトムをそれぞれフロントエンドシステムとバックエンドシステムに分離して格納し、取扱いも、おのおので行なう方式が提案されている。しかし、実際には2つのシステム間の通信にかかる時間は0ではないので、効率を上げるため、アトムの値をとる動作などはバックエンド内部ですませることができるように変更を加えた。その結果前の章で解説した、アトムデータのそれぞれ一部をLFUとフロントエンドにまたがって格納する方式をとっている。2つのシステムの間の通信に必要な処理や、格納方式などについての具体例を以下に示す。

まず第一に、A L P S / IIで実際に使われた、S式読み込みの手順を説明しよう。

◎例、A L P S / IIにおけるS式の読み込み

フロントエンド側にある「読み込み手続」は、バックエンドシステムによって起動されると、その時点で読み込み位置にある文字を見て、まず適当な値を返す。その後、文字が空白、コンマ、左括弧、右括弧、ドット、引用符以外であれば、適当な位置まで読み込みを続けて、「記号アトム」や「文字列」のための文字列を作ったり、数字の列から数値を作ったりしておく。

そしてバックエンドからの要求にしたがい、それらの値へのポインタなどを引き渡す。

A L P S / IIのLISPでは、同じP-nameを持つ記号アトムが2つできることを許していない (文字列でさえも許していない) ので、読み込んだ文字 (列) が、既に記号アトムとして登録されている場合の処理を考えなければならない。フロントエンド側から渡されたハッシュ値と文字数から、バックエンド側でP-nameが同じ可能性のあるものを記号アトム表から探し

、逆にバックエンド側からフロントエンド側にポインタを渡してフロントエンド側で文字列比較をする。そして同一の文字列である場合には、後から定義したポインタを消去する。このような処理方法をとっているので、バックエンド側では文字（列）は必要なく、記号アトム登録のためのハッシュ値などが必要となるだけである。このS式入力時の処理の流れを図4に示す。

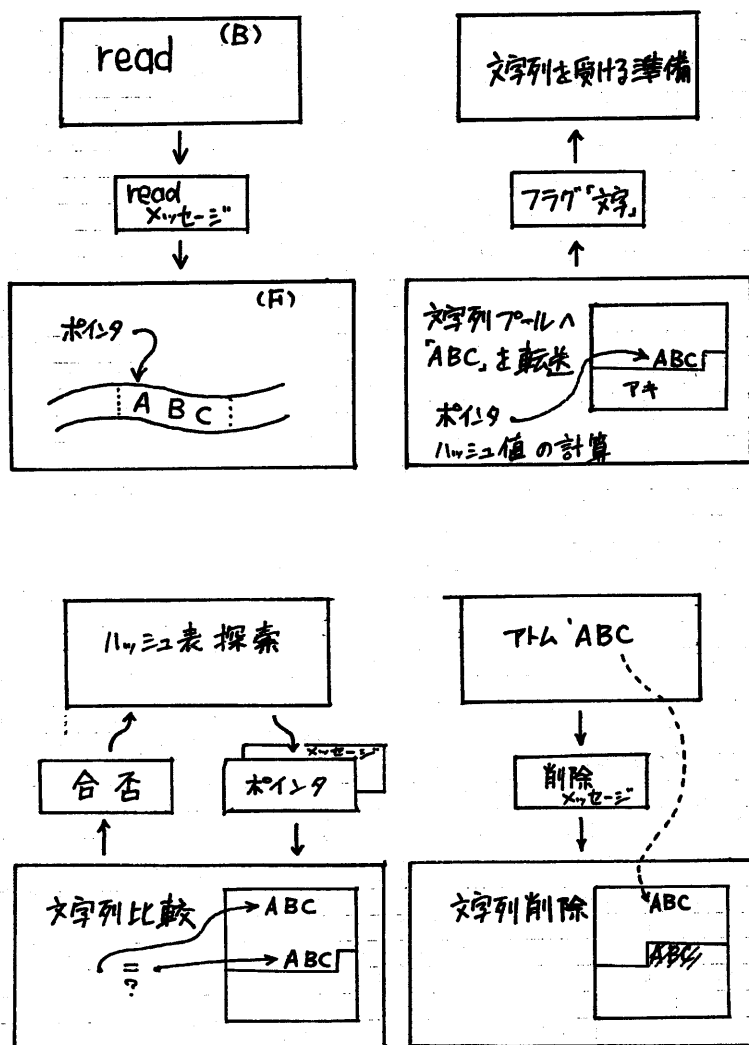


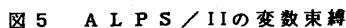
図4 S式入力時におけるフロントエンドとバックエンドの協調動作

◎ 例、EXPLODE、文字列の格納場所の例外

文字列はすべてフロントエンドシステムのメモリ内に取られると書いたが、実際には文字数が1文字と2文字の場合には文字列はフロントエンド側のメモリには取られない。3文字以上の場合には文字数（8ビット）とポインタ（16ビット）が渡されるが、2文字以下の場合には文字数（8ビット）と文字列（16ビット）が渡される。これは文字列を個々の文字に分解するEXPLODEの処理を考えているからである。（EXPLODE 'ABC'）の例では、バックエンドは、

4. 变数束縛

変数束縛の方法は、Shallow-Bindingを採用した。この方法では、変数のアクセスは高速になるが、関数の入口・出口(LAMBDAやPROGの入口・出口)でのオーバーヘッドが『大』になる。そこで、古い値をプッシュ・ダウンし、ポップ・アップするのにスタック(P. D. Stackと名づけたバルクメモリ上にとられるソフトウェアスタック)を用いて、この束縛・開放の処理の能率を上げている。CDRチェーンをたどる必要がないことと、解放したセルがG. C.の対象にならず、すぐ再利用されることによって効率があがる。あたらしい変数束縛によっては、いくつかの変数の名前とそれまでの値(value-cellの値)をP. D. Stackにプッシュ・ダウンする。同時に、そのときまでのP. D. StackのTOSを取って置く。解放のときは、この取って置いたTOSまで、P. D. Stackからポップ・アップし、値を取り換える。この方式をとったのは、通常の変数アクセスと変数の束縛・開放の処理を高速にし、かつFunargの処理も簡便にできるようにするためである。図5は、ラムダ変数Xを持った関数が実行されるとき束縛・開放の様子を示している。束縛時は記号アトムXへのポインタと、古いXの値とをプッシュ・ダウンし、解放時はポップ・アップしている。1回のLAMBDAあるいはPROGなどによる変数束縛によって何という変数(達)が新しくプッシュ・ダウンされたかを示すポインタ(P. D. Stackのフレームポインタ)はLFUのハードウェアスタック上によって実行速度を高める工夫がなされている。



FUNARGの取り扱いにはP・D・Stackを用いて以下のように実現した。

1. FUNCTIONに出あった時、FUNARG形式を作る。その時P・D・StackのTOSP(トップオブスタックポインタ)の値をFUNARG形式につけてやる。

2. FUNARG形式を評価する時は、その時のP・D・StackのTOSPから、FUNARG形式についてきたP・D・Stackへのポインタまでにふくまれる古い値と現在の値とを取り換える。スキャンの方向はTOPからBOTTOMへで、同じ変数が何回出て来ても気にせず交換する(図6)。

3. FUNARG形式から出る時は、スキャンの方向を逆にして交換を行なう。

◎ 処理できるFUNARG形式は

この実現方法では、P・D・Stackの内容が一時的に壊されるので、FUNARG形式は「すなお」な入れ子になっている必要がある。たとえば、Aという函数の中でFUNARG形式が一つ作られる。そしてAから呼ばれた函数でFUNARG形式がもう一つ作られる。このとき、一つのFUNARG形式に入っていて、もう一つのFUNARG形式を評価しようとした時の処理の保証はしない。これを許そうとすると、FUNARG形式を評価中に別のFUNARG形式を評価しようとした時に一度交換した値を元に戻すなどという処理が必要になってくる。ALPS/IIのLISPでは、FUNARG形式を評価中に別のFUNARG形式を定義し、評価することは許すが、外で定義されたFUNARG形式を評価することは考えていない。

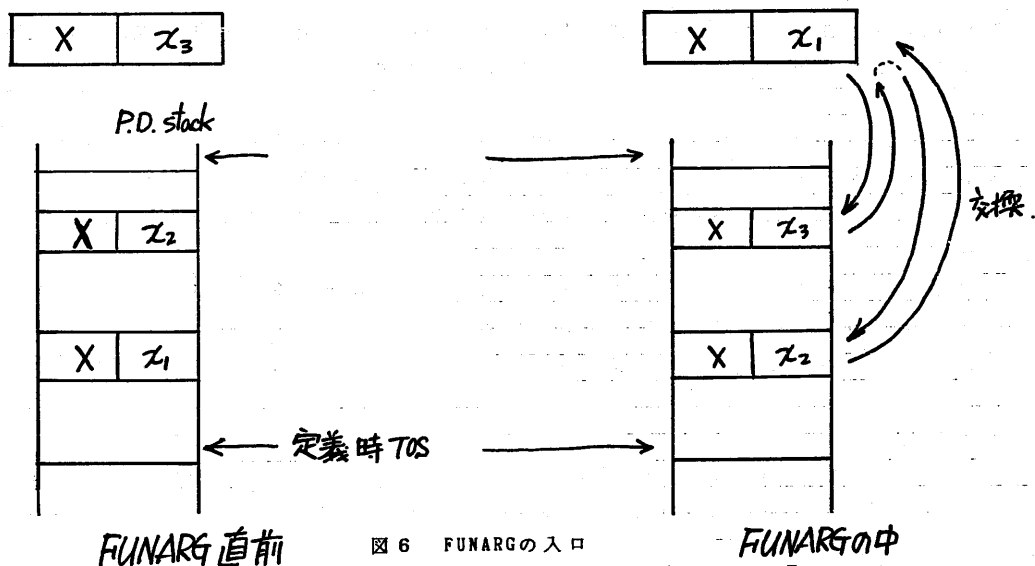


図6 FUNARGの入口

記述するマイクロプログラムの量が増え、アルゴリズムも複雑になってきて、マイクロアセンブラに対する要求も大きくなってきた。プロトタイプ版マイクロアセンブラ[SAT81]に代わって、新しいマイクロアセンブラを設計した。ALPS/IIのLFUのマイクロ命令を生成するアセンブラは、

① データの流れが理解できるものであること、

② 制御している部分が理解しやすい=実際には指定できる機能を使用していない時、それが容易に見つかること(プログラムが短くなる箇所を探しやすいニモニックであること)

③ プログラムの流れが読み取れるものであること、

の3つを満足するものであればよいと考えた。その結果、ALPS/IIのマイクロアセンブラとして、

(1) ソースとALU機能の右に『=>』という記号を書き、さらにその右にディスティネー

ションを書くという文法を取り入れてデータの流を見やすくした。また、ALU動作とスタックプッシュが同時に行える構成になっているので、スタックプッシュを行う場合、先の「=>」の外に「=> PUSH」を書くことにした。

(2) スタックプッシュに対して「=> PUSH」、バルクメモリリードにたいし「B_READ0」、ステータスレジスタ変更に対し「CHG_STAT」などと意味の取り易いキーワードを導入した。

(3) アセンブルリストのうえで出来る限り後続処理指定の条件句におぎないをしてやる。LFUのマイクロ命令は、どの語にも後続処理指定を書くことができる。したがって、この後続処理指定をいかに有効に用いるかは、1つの重要な課題である。この課題は、このおぎないで或る程度解決できると考える。

この考えの下に作成したアセンブラは、間野研究室学部4年生の永栄 繁樹君によって作成され、現在更新中である。それを用いたインタプリタのコーディングの一部を図7に示す。

6. フロントエンドシステム記述ツール

フロントエンドシステムとして、OSにCP/M-86を用いた8086システムを使用している。そのシステムの開発環境を整備する目的で、MDL-FORTRANのモジュラプログラミング・構造化プログラミングの思想を取り入れた高級アセンブラを作成した。ALPS/IIのフロントエンドシステムの一部は、この高級アセンブラを用いて作成した。このアセンブラは、間野研究室修士課程学生の内田 智史君、学部4年榎田 幸君との共同で設計し、作成された。図8にこのアセンブラによるコーディング例を示す。

7. まとめ

第5章で示したマイクロプログラムの例から明らかとなるように、このマイクロアセンブラを用いると、ALPS/II上でのLISPシステムのコーディングは、そのLISPシステムの動作がどのようにして行われるかを第3者に分かり易い形で表わしたものとなる。

8. 謝辞

LFUマイクロアセンブラを作成してくれた永栄 繁樹君、8086開発システム作成に協力してくれた内田 智史君、榎田 幸君に感謝いたします。

なお、本研究において開発した計算機ALPS/IIの作製は文部省科学研究費446193により行なった。

9. 参考文献

1. [IDA79] Ida, M & Mano, K: "An Adaptable Lisp Machine Based on microprocessors." Proc. of Int'l Micro & Mini Computer Conference at Houston. pp210-215, 1979, nov.
2. [SAT81] 佐藤 衛・井田 昌之・間野 浩太郎: LISPマシンALPS/IIの機能とその設計 青山コンピュータサイエンス 第9巻、第2号、pp17-49、1981, Dec.
3. [SAT82a] 佐藤 衛・井田 昌之・間野 浩太郎: 会話型人工知能専用機ALPS/IIのLisp処理機構 情報処理学会記号処理研究会 17-2, 1982, Jan.
4. [IDA82] Ida, M & Mano, K: "Design Considerations on a Lisp System of a Lisp-machine with co-operating processors" in [MAN82] 1982, Mar.
5. [MAN82] 間野 浩太郎: マイクロプロセッサを用いた人工知能・数式処理専用システム 昭和56年度科学研究費補助金(一般研究B)研究成果報告書 1982, Mar.
6. [SAT82b] 佐藤 衛・井田 昌之・間野 浩太郎: ALPS/IIの機能とその設計 理化学研究所シンポジウム、1982, Mar.
7. [SAT82c] 佐藤 衛・井田 昌之・間野 浩太郎: ALPS/IIのfree storage処理機構

```

18      249      C      X+++++++
18      250      C      X
18      251      X      #FUNCTION APPLY ( FN , ARGS )
18      252      C      X
18      253      C      X+++++++
18      254      C      X
19      255      ARG0 * NIL => CHG_STAT ;
20      256      ARG0 => VALR CHG_STAT IF ZERO THEN RETURN ;
21      257      ARG1 => PUSH ;
22      258      ARG0 => PUSH IF NOT_ATOM THEN JUMP_TO APPLY_NOATOM ;
22      259      C      X
23      260      !EXPR1 => ARG1 RETADR => PUSH ;
24      261      CALL GET-SUB ;
25      262      IF NOT.TRUE THEN JUMP_TO APPLY_NOEXPR ;
26      263      IF TRACE THEN VAL3 => ARG0 VALR => PUSH
27      264      ELSE JUMP_TO APPLY_NOTR_EXPR ;
28      265      STACK ( TTOS + 4 )=> ARG1 ;
29      266      CALL TRACE-ENT-PRN ;
30      267      STACK ( TTOS + 4 )=> ARG1 ;
31      268      STACK ( TTOS + 3 )=> ARG0 ;
32      269      CALL TRACE-ARG-PRN ;
33      270      STACK ( TTOS + 4 )=> ARG1 ;
34      271      POP => ARG0 CALL APPLY ;
35      272      STACK ( TTOS + 3 )=> ARG0 ;
36      273      VALR => ARG1 VALR => PUSH CALL TRACE-EXT-PRN ;
37      274      POP => VALR ;
38      275      APPLY-RETURN ;

```

<< ALPS/II CROSS ASSEMBLER >>

```

LINE#      M      SOURCE STATEMENT
39      276      POP => RETADR ;
40      277      TOSP + '2' => TOSP RETURN ;
40      278      C      X
41      279      APPLY_NOTR_EXPR :
42      280      STACK ( TTOS + 3 )=> ARG1 ;
43      281      VALR => ARG0 CALL APPLY ;
44      282      JUMP_TO APPLY_RETURN ;
44      283      C      X
45      284      APPLY_NOEXPR :
46      285      !SUBR1 => ARG1 ;
47      286      VAL3 => ARG0 CALL GET-SUB ;
48      287      IF NOT.TRUE THEN JUMP_TO APPLY_NOSUBR ;
49      288      IF TRACE THEN VAL3 => ARG0 VALR => PUSH ELSE JUMP_TO APPLY_NOTR_SUBR ;
50      289      STACK ( TTOS + 4 )=> ARG1 ;
51      290      CALL TRACE-ENT-PRN ;
52      291      STACK ( TTOS + 4 )=> ARG1 ;
53      292      STACK ( TTOS + 3 )=> ARG0 ;
54      293      CALL TRACE-ARG-PRN ;
55      294      STACK ( TTOS + 4 )=> ARG0 ;
56      295      POP => JMPINDX ;
57      296      CALL ( JMPINDX ) ;
58      297      STACK ( TTOS + 3 )=> ARG0 ;
59      298      VALR => ARG1 VALR => PUSH CALL TRACE-EXT-PRN ;
60      299      POP => VALR JUMP_TO APPLY_RETURN ;
60      300      C      X
61      301      APPLY_NOTR_SUBR :
62      302      STACK ( TTOS + 3 )=> ARG0 ;
63      303      VALR => JMPINDX ;
64      304      CALL ( JMPINDX ) ;
65      305      JUMP_TO APPLY_RETURN ;
65      306      C      X
66      307      APPLY_NOSUBR :
67      308      VAL3 => ARG0 CALL SYS_ASSOC ;
68      309      IF NOT.TRUE THEN JUMP_TO APPLY_ERR_A9 ;
69      310      STACK ( TTOS + 2 )=> ARG0 ;
70      311      STACK ( TTOS + 3 )=> ARG1 ;
71      312      CALL APPLY ;
72      313      JUMP_TO APPLY_RETURN ;
72      314      C      X
73      315      APPLY_ERR_A9 :
74      316      STACK ( TTOS + 2 )=> ARG2 ;
75      317      !APPLY! => ARG1 ;
76      318      !A9! => ARG0 ;
77      319      CALL PRINT_ERROR ;
78      320      NIL => VALR JUMP_TO APPLY_RETURN ;
78      321      C      X
79      322      APPLY_NOATOM :
79      323      C      X
79      324      C      X STACK ( TTOS + 1 )=> ARG0 ; FN
79      325      C      X STACK ( TTOS + 2 )=> ARG0 ;
79      326      C      X
80      327      IF B_COMD_RDY THEN ARG0 => B_ADRO RETADR => PUSH B_READO
81      328      ELSE WAIT ;
82      329      IF B_DATA_RDY THEN B_CARO * 'LAMBDA' => CHG_STAT
83      330      ELSE WAIT ;

```

図7 マイクロアセンブラによるAPPLY (1部)

LINE#	M	SOURCE STATEMENT
84	331	IF ZERO THEN JUMP_TO APPLY_CARLAMBDA ;
85	332	B_CARO ← FNARG1 ⇒ CHG_STAT ;
86	333	IF ZERO THEN JUMP_TO APPLY_CARFNARG ;
87	334	STACK (?TOS + 2) ⇒ ARG0 ;
88	335	CALL EVAL ;
89	336	STACK (?TOS + 3) ⇒ ARG1 ;
90	337	VALR ⇒ ARG0 CALL APPLY ;
91	338	JUMP_TO APPLY_RETURN ;
91	339	C X
92	340	APPLY_LAMBDA1
92	341	C X
92	342	C X STACK (?TOS + 1) : RETADR
92	343	C X STACK (?TOS + 2) : ARG01 FN
92	344	C X STACK (?TOS + 3) : ARGS
92	345	C X B_CARO1 CAR FN
92	346	C X B_CDRO1 CDR FN
92	347	C X
93	348	IF B_CMD_RDY THEN B_CDRO ⇒ B_ADRO B_READO
94	349	ELSE WAIT ;
95	350	IF B_DATA_RDY THEN B_CDRO ⇒ PUSH
96	351	ELSE WAIT ;
97	352	STACK (?TOS + 4) ⇒ ARG1 ;
98	353	B_CARO ⇒ ARG0 CALL ARGS_PUSHDOWN ;
99	354	IF B_CMD_RDY THEN POP ⇒ B_ADRO B_READO
100	355	ELSE WAIT ;
101	356	IF B_DATA_RDY THEN B_CARO ⇒ ARG0 CALL EVAL
102	357	ELSE WAIT ;
103	358	VALR ⇒ PUSH CALL ARGS_POPUP ;
104	359	POP ⇒ VALR JUMP_TO APPLY_RETURN ;
104	360	C X
105	361	APPLY_FNARG1
105	362	C X
105	363	C X STACK (?TOS + 1) : RETADR
105	364	C X STACK (?TOS + 2) : ARG01 FN
105	365	C X STACK (?TOS + 3) : ARGS
105	366	C X B_CARO1 CAR FN
105	367	C X B_CDRO1 CDR FN
105	368	C X
106	369	IF B_CMD_RDY THEN B_CDRO ⇒ B_ADRO B_READO
107	370	ELSE WAIT ;
108	371	IF B_DATA_RDY THEN B_CARO ⇒ PUSH
109	372	ELSE WAIT ;
110	373	B_CARO ⇒ ARG0 CALL ARGS_SAVE ;
111	374	STACK (?TOS + 4) ⇒ ARG1 ;
112	375	POP ⇒ ARG0 CALL APPLY ;
113	376	VALR ⇒ PUSH CALL ARGS_RESTORE ;
114	377	POP ⇒ VALR JUMP_TO APPLY_RETURN ;

<< 61>> MODULE GET_ONE_REC

図7 マイクロアセンブラによるAPPLY (1部)

```

1289 : #PROC GET_ONE_REC
1290 : : MOV BX,0FFFFH
1291 : : MOV SI,80H
1292 : : MOV DI,OFFSET ACC_BUF
1293 : : ADD DI,BP
1294 : : MOV AL,[SI]
1295 : : *IF BP = 0H THEN
1296 : : : *EXECUTE< 64> SKIP_BLANK_SYM
1297 : : : *END-IF
1298 : : : *LOOP GET_LOOP
1299 : : : *IF E_FLAG ≠ 0H THEN
1300 : : : : *REPEAT GET_LOOP2
1301 : : : : : <---*LEAVE GET_LOOP2 ( AL < 20H )
1302 : : : : : MOV [DI],AL
1303 : : : : : INC DI ! INC SI ! INC DX
1304 : : : : : *IF SI >= 100H THEN
1305 : : : : : SUB DI,OFFSET ACC_BUF
1306 : : : : : MOV BP,DI
1307 : : : : : *LEAVE GET_LOOP
1308 : : : : : *END-IF
1309 : : : : : MOV AL,[SI]
1310 : : : : : *UNTIL AL < 20H DO GET_LOOP2
1311 : : : : : *EXECUTE< 62> CMP_STRINGS
1312 : : : : : *IF Z2 = 0H THEN
1313 : : : : : *EXECUTE< 63> GET_ADR_BX
1314 : : : : : MOV E_FLAG,1
1315 : : : : : *LEAVE GET_LOOP
1316 : : : : : *ELSE
1317 : : : : : : MOV DI,OFFSET ACC_BUF
1318 : : : : : : MOV DX,0H
1319 : : : : : : *EXECUTE< 64> SKIP_BLANK_SYM
1320 : : : : : *END-IF
1321 : : : : : MOV BP,0H
1322 : : : : : *LEAVE GET_LOOP
1323 : : : : : *END-IF
1324 : : : *END-LOOP GET_LOOP
: : : *END

```

図8 CP/M-86 用高級アセンブラのコーディング例

0 ERRORS IN THIS MODULE