論　文

ミニコンにおける APL 会話型処理システム*

渡辺豊英** 宮脇冨士夫***
渡辺勝正** 萩原　宏**

Abstract

This paper deals with the presentation of APL interpreter implemented on a mini-computer (HITAC-10). It includes a syntax analyzer which translates the external APL language to an intermediate language. The analysis phase consists of two steps in order to distinguish unary operators from binary operators and so on. Firstly it scans from left to right and then from right to left. Moreover during the execution of all APL statements dynamic allocation is used. In the matter of having no sufficient data areas in main memory, this system makes use of kinds of data management techniques.

We describe the processing method of APL interpreter, discuss how wide range of faculty the interactive processing system on a mini-computer covers, and also show the design concept to make APL matched with a mini-computer's characteristic.

1. まえがき

APL は，1962年 K. E. Iverson によって発表された言語にもとづくもので[1),2)]，当初は，計算機のプログラミングを対象とした言語ではなく，アルゴリズムを簡潔に表現することを目的とした記述能力の大きい言語であった[3),4)]．この言語のプログラミング言語への適用は，1964年 H. Hellerman の実験会話型システムがあり[5)]，さらに，IBM system/360 に対してインプリメントされた APL\360 などがある[6)]．今日，一般に APL と言えば，プログラミング言語として使われているものである．

記述言語とプログラミング言語では，表現形式に多少の相異はあるものの，アルゴリズム記述の容易さという点では，まったく同じである．後者はタイプライタで入力可能な一次元表現をとり，人間と機械のコミュニケーションの一方法という観点から改良されている．

APL は，視覚的・感覚的な要因を含み，従来の FORTRAN に代表されるプログラミング言語とは外形が異なる[7)]．プログラミング言語として，アルゴリズムが記述し易いという長所は，十分な要素である．APL 会話型プログラミング言語の特徴を述べれば，次のようである．

i）アルゴリズムの表現が簡潔，かつ明瞭である．
ii）演算子の種類が多く，その表現が視覚的で簡明である．
iii）変数の型について，使用者はまったく意識する必要がない．（変数を宣言する必要がない．）
iv）演算子を定義することができ，かつそれを修正することができる．（関数を定義できる．）
v）構文の構造が，演算子と演算数の組合せから成り立っている．

我々は，このような特徴をもつ言語を実際に利用できるようにすることに興味をもち，ミニコンにおけるその処理システムの作成を試み[8),9)]，完成させた[10)]．インプリメントにあたっては，APL をミニコンの特性に一致させることを主眼とした．

この論文では，前述の APL の特徴を実現するため

* APL Interactive Processing System On A Minicomputer By Toyohide WATANABE, Katsumasa WATANABE, Hiroshi HAGIWARA (Department of Imformation Science, Kyoto University) and Fuzio MIYAWAKI (Himeji Institute of Technology)
** 京都大学工学部情報工学教室
*** 姫路工業大学

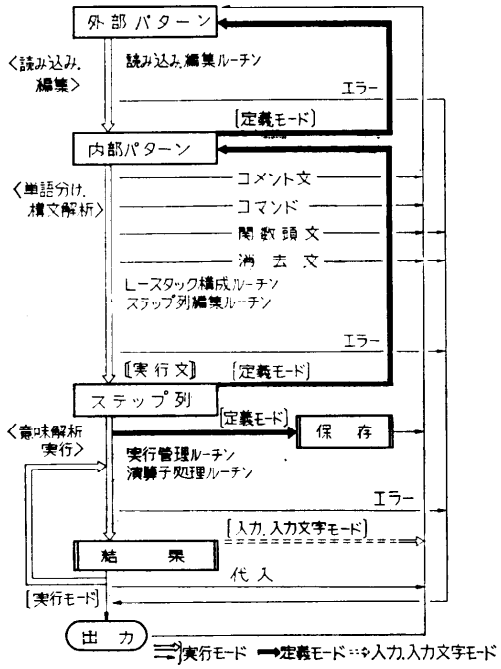

Fig. 1 Flow of processing Data

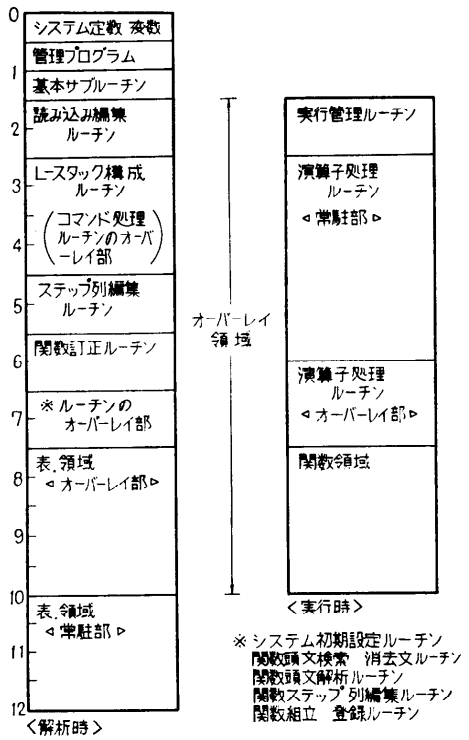

Fig. 2 Memory Map

の処理方法と，ミニコンにおける会話型システムの構成を述べるとともに，当システムで明らかになった問題点を示す．

2. システムの概要

2.1 処理方法

当システムの構文は，**付録 I** (p. 787 参照) の言語仕様による．処理方式は，インタープリター方式で，原文をステップ列と呼ぶ中間言語に変換し，それを順次実行してゆく．処理データの流れは，**Fig. 1** の如く実行モードと定義モードにわかれる．実行モードでは，中間言語の生成後実行し，定義モードでは，中間言語を保存する．また，原文は，読み込み・編集，単語分け・構文解析のトランスレータ部と，意味解釈・実行のインタープリター部にわかれる．

APL では，変数の型・寸法を宣言する必要がなく，ダイナミックにその型・寸法を変えることができる．従って，変数の性質は実際に値が与えられるまで確定しないので，構文解析の段階で演算数の型を決定できず，機械語に翻訳できない．しかし，使用者が定義した関数の実行においては，原文から直接処理するのでは効率が悪いので，中間形式のステップ列に変換された言語を実行する方法をとった．

2.2 システムの構成

当システムは，HITAC-10 (12 kW)，磁気ドラム (131 kW) と APL タイピングエレメントをもつタイプライタで構成される．

処理プログラムのステップ数は，2 万語 (1 語 16 ビット) を越える．さらに，必要な表・領域を含めて，コアメモリ上に全て常駐することができず，ドラムを使ってオーバーレイを行う．コアは，**Fig. 2** の如く構文解析時と実行時の 2 つの様相を示す．すなわち，処理方式が中間言語の生成と実行のフェーズで大別され，実行効率をたかめるために割付けも，これを支える構成とした．さらに，処理プログラムの半分を占める演算子処理ルーチンは，使用頻度の多いと思われるものを常駐部とそれ以外のものを 5 グループのオーバーレイ部にわけた．

2.3 システムの設計方針

当システムの特徴を与える設計方針を簡潔に列挙する．

i) 実行効率がたかい構成とする
ii) ミニコンの特性を充分に生かす
iii) プログラム構造は簡単な構成とする

ミニコンが特定の場で，限られた問題に使われ，個々の使用者にファイルを与える必要がなく，共用ファイルの構成とした．さらに，コマンドをできる限り少くして，コマンドを意識することなく使える．一方，デバッギングに利点があるように，2種のコマンドをもうけた．当システムは，メモリの制約もあって割愛した機能もあるが，APL/360 と比較してもそれ程劣ることがない．割愛の機能は，配列を2次元まで，それに伴う演算子の縮小である．また，各演算子処理ルーチンでは，共通な処理部分があり，コアの有効使用をはかるようにプログラムの構成とした．

3. 中間言語の生成

3.1 APL 言語の構文構成の特徴

構文からの APL 言語の特徴を述べることは，以下の中間言語生成過程で必要なことである．

ⅰ）演算の順序は，右から左へと連続的に行なう．ただし，カッコの中は，1つの演算数とみなす．

ⅱ）演算子に順位がない．

ⅲ）演算子の数が多く，単項・2項の両方に使用されるものも多い．

ⅳ）変数の宣言がない．

ⅴ）出力の有無は，文型によって決る．

さらに，構文構成が演算子と演算数の関係で成り立ち，右から左への演算において，演算子の機能はその演算子の左演算数と右演算式に作用すると考えることができる．単項の場合は右演算式のみに作用する．それ故に，演算子個々には順位がない．

3.2 変換過程

原文は，読み込み・編集のフェーズで，内部コードの並び列である内部パターンへと変換される．ここでは，重ね打ち記号の処理とタイプイン記号の検証を行う．

次に，単語分け・構文解析のフェーズでは，内部パターンからステップ列（基本形は，Quadruple）へと変換する．APL の演算順序は，“右から左へ”であるから，走査も右から左へと行なう必要がある．しかし，単語分けでは左から右へと走査する方が，右から左へと走査するより容易であり，さらに演算子の単項・2項の区別は左演算子数の有・無で決るので，一度の走査では記号の先読みが必要で複雑になるので，2段階の処理で変換する．すなわち，“左から右”の走査で，単語分けと演算子の単項・2項の区別をし，“右から左”の走査で演算順序の決定を行なう．

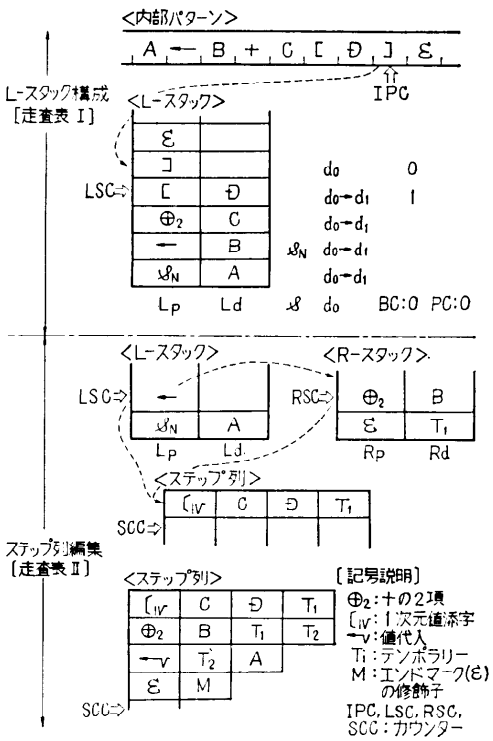

Fig. 3 Flow of Syntax Analysis from Innerpattern to Step-Sequences

3.2.1 左から右への走査

内部パターンを**走査表 I**（**付録 II**）(p. 788 参照)に従って，Lスタック（演算子と演算数の組を2トラックに記憶するスタック（**Fig. 3**））に組み込んでゆく．内部パターン上で走査中の記号とLスタックの最上段にある演算子の組合せによって，動作が決る．また，内部パターンのコード列をエントリポインタと呼ぶ表の出口の番地に変換する．演算子は演算子表の，変数名・関数名はそれぞれの名前表の，定数は定数表のポインタである．

このステップの走査内容は，上記以外に

ⅰ）単項・2項の演算子の区別

ⅱ）文型の決定（出力の有・無）

ⅲ）(と)，[と] のレベルカウント

ⅳ）構文の検証

などである．

単項・2項の区別は，Đフラグという指示フラグで行う．すなわち，次の演算子が，単項・2項かを示す2状態フラグである*．走査表 I の動作は，全てこの

* 走査表 I の d_0, d_1 がそれにあたる．d_0 は次が単項であることを，d_1 は次が2項であることを示すとともに構文検証をもする．

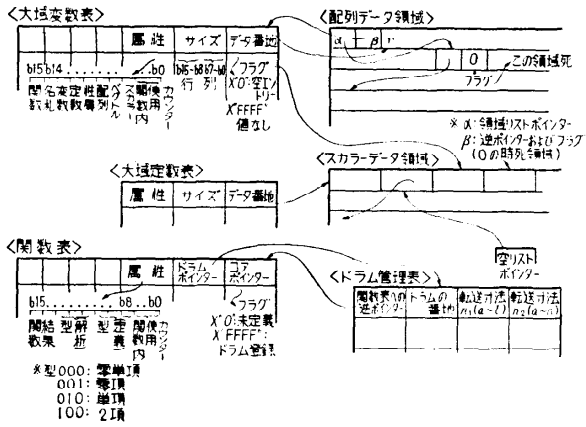

Fig. 4 Data Entries

Đ フラグの判別から初め，Đ フラグの設定で終る．

APL では宣言がないので，変数登録ではその名前だけの登録を行い，属性・サイズのデレクトリは実行まで検証しない．また，このデレクトリは実行時にダイナミックに変化できる構造である．(**Fig. 4**)．

3.2.2 右から左への走査

Lスタックに組み込まれたエントリポインタ列を**走査表 II (付録 III)** (p. 788 参照) に従い，補助的にRスタック (構造はLスタックと同形である．) を用いてステップ列に変換する．

このステップの主な走査は，

i) ステップ列への変換

ii) 先のステップで確定しなかった構文の決定

である．

走査表 II の動作は，LスタックとRスタックの最上段の演算子の組合せによって選択される．すなわち，

イ) $L_p \geqslant R_p$ の時*

a) スタックの L_p 部によってステップ列を形成する．(単項演算子の場合)

b) Lスタックの内容の一部をRスタックへ移す．(2 項演算子の場合)

ロ) $L_p < R_p$ の時

Rスタックの R_p 部によってステップ列を形成する．

ハ) $L_p \doteqdot R_p$ の時

イ) または，ロ) の動作を行なう．

* L_p 部と R_p 部の演算子に伴うステップ生成の順位である．また $\geqslant$ $\doteqdot$ $<$ 記号はその比較を表す．

である．

3.3 中間言語処理について

このように 2 つのステップを経ても，完全に解析できないことがある．それは，未定義関数の型を決定することで，この場合にはある程度不明確なまま実行段階に移り，関数の実行に際して型を参照し，必要ならば関数呼びだしのステップ列を変形する．これは，APL では関数の型の宣言がなく，定義されて初めて決定できることによる．しかし，関数を内部パターンのまま保存しておいて，実行する直前にステップ列に変換する方法をとれば，避けられることであるが，我々は実行効率をたかめるため，ステップ列の形で保存する方法をとった．さらに，早期に関数の構文を検証することによって無駄な登録を避けることもある．

一方，この方法による問題点を述べれば，

i) 再定義などの場合，もう一度中間言語を生成するという重複する部分がある．

ii) 中間言語から原文を復元することが困難である．

iii) 名前が一度解析されると，変数・関数に決定されるので，名前の自由な移行に多少の難点がある．

であり，ii) については，関数の登録保存構造に，ステップ列と内部パターンを持つことにした．iii) では，変数から関数の移行に関して，システムでその検証・変換を行なう．

4. データの扱いとその領域

4.1 データ領域の扱い

変数は，ダイナミックにその属性・サイズを代えることができるので固定領域を保持することができない．従って，限りある領域を最大に利用するための工夫が必要である．

演算が実行される場合，各演算子処理ルーチンは，その属性・サイズを検証し，演算結果のための領域を確保する．実行の途中で中間結果の記憶に使われたデータ領域を再使用可能か否かを調べ，可能なら解除してデータ領域の使用可能な部分を常に最大にしてから領域を確保する．また，エンド・マーク処理ルーチンでは，各演算子処理ルーチンで解除し得なかった領域を解除する．定数等がこれにあたる．

APL では，会話の途中で定数が与えられたり，ベクトル・配列の構造をした定数が許されるので，定っ

た定数表を用意して，1つの定数に同一ポインタを与えることができない．そのためにプログラムで使用される度に定数を作る．実行が終れば不要なものである．

さらに，データ領域管理ルーチン内に，ガーベージコレクタを組み込んで，連続した空領域が小さいとき（フラグメント化）には，分散している使用領域をつめて大きな空領域を作る．

4.2 データ領域の構造

領域全体の有効性をはかる以外に，データ領域それ自身の有効性をたかめるために，関数内データ領域と大域データ領域とにわけた．関数内で使用される定数は，それが実行されるときのみ必要であり，関数の構造* に持っていて実行時にコアの特定の領域にロードするという手法をとった．これは，定数編集を簡単にし，さらに領域の有効利用にメリットがあった．

一方，大域データ領域を，スカラ領域と配列領域**にわけた．その理由は，APL が配列を扱うのにすぐれてはいるが，スカラの使用頻度が大きいと予想されるからである．スカラ領域は，2語セルが全てリスト構造をとっている．また，配列領域は，不定長のセルが2語の情報セルによってリスト構成化されている (Fig. 4)．この情報セルは，空か否かにかかわらずチェインされていて，1語はこのチェインのポインタであり，他の語はその領域が空か否かを示す．空では空指示を，それ以外では，エントリの逆ポインタとなる．ガーベージコレクタは，この逆ポインタで領域の再構成をして，データポインタを書き換え，その結果大きな領域を作る．

4.3 データ形式

データには，数値と文字がある．数値は，システム内では全て浮動小数点で，2語長で表現される．入力データ形式は，フリーフォマットであるが，出力されるときは，数値によって固定小数点か，浮動小数点かが選ばれる．一方，文字は，1記号1語で表現される配列データとして扱う．

4.4 名札の扱い

APL の分岐には，分岐記号の演算数として，名札，定数，変数（演算した結果も含む）の3種類がある．それぞれに扱っては複雑であるので，名札を構文解析で定数と同一レベルの扱いができるように変換する．すなわち，データエントリ構造を3種とも同一とする．

5. ミニコンでのシステム構成

5.1 共用ファイル

当システムの設計対象は，タイムシェアリングでの複数の使用者ではなく，単一処理での複数の使用者である．従って，ドラム上のファイルを共用ファイルとし，使用者のファイルに対する知識を一切排除して，作成された関数を自動的に登録する．また，その消去も特定の制御をすることなく，APL 言語内で行なえることは，APL の特徴と一致する．

共用ファイル採用の利点は，

ⅰ）登録された関数は全て共用できる．

ⅱ）システム構成が容易であり，かつ使用者も扱い易い．

であるが，一方，機密機構をもたないので，他の使用者によって破壊される場合がある．しかし，当システムでは，個人ファイルとして磁気テープを使用できる．

5.2 共用ファイル構成と中間言語からの実行で生ずる問題

当システムでは，実行効率をたかめるために定義された関数を中間言語から実行するが，この中間言語の各ポインタ列は，コア内のある絶対番地へのポインタであるので，エントリが消去文（▽{名前}*▽）でシステムから抹消された場合，名前が存在しないのにポインタのみが生きているという状況が起ることがある．このような場合を防ぐために，当システムでは，関数内使用カウンターを各エントリ内にもうけている．以下，その手続きを述べる．

ⅰ）ある関数が定義された場合

その関数で使用する大域変数名と他の関数名を調べる．次に，使用されている名前はそれぞれのエントリ内の関数内使用カウンターが1つインクリメントされる．すなわち，その名前がいくつの関数で使用されているかをカウントするのが，このカウンターの役割である．

ⅱ）ある名前が消去されようとした場合

その名前のカウンターが，カウントされているか否かを調べる．次に，カウントされていない場合は，消去操作をする．関数であれば，さらにその関数が使用している名前のカウンターを1つデクリメントする．一方，カウントされている場合は，消去しないというメッセージを送る．

---

* 関数をファイルに登録する場合，定義された関数そのままの形ではなく，文番表・ステップ列・関数内データ・内部パターン等で構成されたものである．これを関数の構造と呼ぶ．

** スカラとは，数値の1個データのことを言い，そのための領域をスカラ領域といい，これ以外を配列領域にデータを格納する．

デットロックに落ち入らないように，消去指定を複数個とした．さらに，コマンドの使用がある．このようなオーバーヘッドは，実行時のそれとは異るので，システムの実行効率を害するものではない．

5.3 使用者の実行制御について

ミニコンは，大型機に比べて使用者と密接に依存している．すなわち，実行中に割込み後デバッギング情報を求めたり，実行中にデータを与えて何度も検討するという専有的な使い方に便利でなければならない．

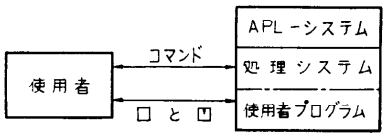

Fig. 5 Communication between User and System

Table 1 List of Commands.

コマンド	機　　能
CLEAR	システムの初期化
VARS	登録大域変数のリスト
FUNS	登録関数のリスト
DIGIT	有効数字の桁数指定
TRACE	関数の一文の計算結果の出力
STOP	関数の一文の停止の指示
LIST	中断している関数のリスト
RESTART	中断している文から実行指示
RETURN	中断している関数を呼んだ文からの実行

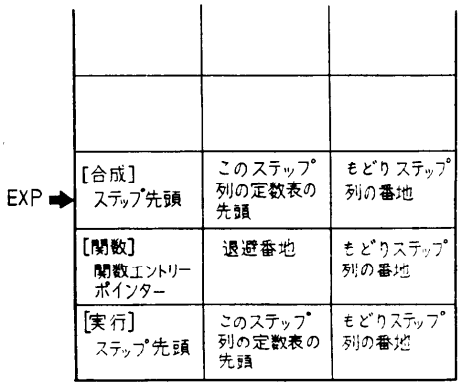

Fig. 6 State of Control Management in Execution Stack. This case is firstly function call, secoundly mixed operator evaluation and other evaluation

* 1トラック目は関数の場合にはそのエントリーポインターで，それ以外の時はステップ列の先頭である．2トラック目は，関数の場合は局所変数等の退避番地であり，それ以外はこのステップ列の定数表の先頭番地である．3トラック目はもどり番地である．

さらに，APL 言語がこの特徴によく一致する．**Fig. 5** の如く使用者が計算機に対して，直接意志を伝えるには2つの方法がある．一つは，□と⊡を使用してプログラムと会話する方法であり，もう一つは，コマンドによって処理システムに指令を与える方法である．この2つの機能によって，デバッギングが容易となる．さらに，当システムでは特にデバッギングのためのコマンドとして，RESTART, RETURN の2つを加えて実行段階で使用者が，プログラムの流れを制御することを可能にした（**Table 1**）．

実行制御の管理は，3トラック* の実行スタックで行ない，関数の再帰呼びだし等，実行の現時点までの回復は記憶内容を用いて行なう（**Fig. 6**）．

6. あとがき

APL の特徴をミニコンの特性に適合させた会話型処理システムの構成結果を述べたが，領域の割付け等のいくつかの問題が残されている．たとえば，データ領域も演算子処理ルーチンに使用したオーバーレイ方式とすることで処理量を増大させることができる．

また，当システムでは，実行効率をたかめるために，関数は中間言語の形で記憶されて実行される方法を採用した結果，共用ファイルの問題，名前の関数から変数への移行等に，多少の制限を加えた．しかし，実行効率と動的特性，および領域の割付け方などから検討しなければならない内容である．

当システムの設計方針であった項目については，一応満足する結果を得たと思われるが，将来，計算機の処理能力が向上したり，システム構成に適合するハードウェアユニットが付加された場合には，さらによいシステム構成をなし，効率のたかい結果が得られる．たとえば，ダイナミックマイクロプログラム方式の計算機の利用[11),12)]や，配列処理の機構[13)]の利用などが報告されている．これらの方面の研究をすることによって，APL 言語の特徴がさらに生かされると思われる．

最後に，システム開発に要したマンパワーは6人年であったことを報告すると共に，このシステムの作成に協力された松本雅史，増田康男両君をはじめ萩原研究室の方々に感謝する．

参 考 文 献

1) Iverson, K. E.: A Programming Language, Proc. AFIPS 1962 SJCC, Vol. 21, pp. 345～351.

2) Iverson, K. E.: A Programming Language, p. 286, John Wiley & Sons. Inc., New York (1962).
3) Falkoff, A. D.: Algorithms for Parallel-Search Memories, J. ACM, Vol. 9, No. 4, pp. 488～511 (Oct. 1962).
4) Falkoff, A. D., K. E. Iverson & E. H. Sussenguth: A Formal Description of System/360, IBM System Journal, Vol. 3, No. 3, pp. 193～262 (1964).
5) Hellerman, H.: Experimental personalized array translator system, C. ACM, Vol. 7, No. 7, pp. 433～438 (July 1964).
6) Berry, P. C.: APL/1130 Primer, p. 220, IBM Corp., New York (1968).
7) Falkoff, A. D. & K. E. Iverson: The Design of APL, IBM J. Res. Develop., 17, pp. 324～334 (1973).
8) 宮脇他: APL 会話型処理プログラム (1) そのシステム構成とデータの取扱い, 第 14 回情報処理学会大会予稿 43, pp. 85～86 (1973).
9) 渡辺他: APL 会話型処理プログラム (2) 構文解析の手順, 第 14 回情報処理学会予稿 44, pp. 87～88 (1973).
10) 渡辺他: ミニコンにおける APL 会話型処理システムのインプリメンテーション, 第 15 回情報処理学会大会講演論文集 223, pp. 445～446 (1974).
11) Hasitt, A., J. W. Lageschulte & L. E. Lyon: Implementation of a High Level Language Machine, C. ACM, Vol. 16, No. 4, pp. 199～212 (April 1973).
12) Zaks, R., D. Steingart & J. Moore: A firmware APL time-sharing system, Proc. AFIPS 1971 SJCC, Vol. 38, pp. 179～190.
13) Thurber, K. J. & J. W. Myrna: System Design of a Cellular APL Computer, IEEE Trans. Computers, Vol. c-19, No. 4, pp. 291～303 (April 1970).

(昭和50年 3 月31日受 付)
(昭和50年 5 月 1 日再受付)

付録 I-1 言語仕様

超言語形式で記述する．その約束は以下の如く

1. BNF 記述を基本とする
2. $\{\ \}_a^b$ の意味は以下の様である．

a \ b	j	なし
i	i～j	i
なし	0～j	1

3. APL 記号で表現される．
4. APL 記号以外の表現は．

cr: carriage return　␣: 空白で 1 個以上

n: 自然数　*i*, *j*: $0 \leq i+j \leq 6$ の自然数

付録 1-2

```
<PROGRAM>::=<FUNCTION BLOCK>|<STATEMENT>|<COMMAND>
<STATEMENT>::=<COMMENT LINE>|<EXECUTION LINE>
           |<ALTERATION LINE>
<FUNCTION BLOCK>::=<FUNCTION HEAD LINE>cr<FUNCTION BODY>
           cr<FUNCTION TAIL LINE>|<DELETION LINE>
<FUNCTION HEAD.LINE>::= ∇{<DEFINITION HEAD>
           |<ALTERATION HEAD>}
<FUNCTION TAIL LINE>::= ∇
<DEFINITION HEAD>::=<VARIABLE> ← <SPECIFIED 0-ARG FUNCTION>
            {<PARAMETER LIST>}^1 |<VARIABLE> ←
            <SPECIFIED 1-ARG FUNCTION>␣<VARIABLE>
            {<PARAMETER LIST>}^1 |<VARIABLE> ← <VARIABLE>␣
            <SPECIFIED 2-ARG FUNCTION>␣<VARIABLE>
            {<PARAMETER LIST>}^1
           |<UNSPECIFIED 0-ARG FUNCTION>{<PARAMETER LIST>}^1
           |<UNSPECIFIED 1-ARG FUNCTION>␣<VARIABLE>
            {<PARAMETER LIST>}^1 |<VARIABLE>␣
            <UNSPECIFIED 2-ARG FUNCTION>␣<VARIABLE>
            {<PARAMETER LIST>}^1
<PARAMETER LIST>::={ ; <LOCAL VARIABLE>| : <FUNCTION>}
           |<PARAMETER LIST>{ ; <LOCAL VARIABLE>
           | : <FUNCTION>}
<FUNCTION>::=<SPECIFIED 0-ARG FUNCTION>
           |<UNSPECIFIED 0-ARG FUNCTION>
           |<SPECIFIED 1-ARG FUNCTION>
           |<UNSPECIFIED 1-ARG FUNCTION>
           |<SPECIFIED 2-ARG FUNCTION>
           |<UNSPECIFIED 2-ARG FUNCTION>
          ::=<NAME>
<LOCAL VARIABLE>::=<NAME>
<ALTERATION HEAD>::=<FUNCTION>
<FUNCTION BODY>::={<STATEMENT>|<COMMAND>}|<FUNCTION BODY>
           cr{<STATEMENT>|<COMMAND>}
<DELETION LINE>::= ∇{<NAME>}^n ∇
<COMMAND>::= ) <COMMAND WORD>
<COMMAND WORD>::= LIST | FUNS | VARS | CLEAR | RETURN | RESTART
           |DIGIT␣m|{TRACE | STOP}␣<FUNCTION> ←
           {␣<STATEMENT NUMBER>|␣}^n
<COMMENT LINE>::= ⍝ <CHARACTER STRING>
<CHARACTER STRING>::=<CHARACTER>|<CHARACTER STRING><CHARACTER>
<CHARACTER>::=<LETTER>|<DIGIT>|<SYMBOL>
<ALTERATION LINE>::=<STATEMENT NUMBER>{<EXECUTION LINE>| ∇}
<EXECUTION LINE>::={<LABEL> :}^1{<BRANCH>|<SPECIFICATION>
           |<FUNCTION LINE>|<IMMEDIATE>}
<BRANCH>::= →{␣|<EXPRESSION>|<LABEL>}
<SPECIFICATION>::={<VARIABLE>|⎕} ← <EXPRESSION>
<IMMEDIATE>::=<EXPRESSION>|<IMMEDIATE> ; <EXPRESSION>
<FUNCTION LINE>::=<UNSPECIFIED 0-ARG FUNCTION>
           |<UNSPECIFIED 1-ARG FUNCTION>␣<EXPRESSION>
           |{<OPERAND> | ( <EXPRESSION> )}␣
           <UNSPECIFIED 2-ARG FUNCTION>␣<EXPRESSION>
```

付録 1-3

```
<EXPRESSION>::=<OPERAND>|<MONADIC OPERATOR><EXPRESSION>
           |( <EXPRESSION> ) <INDEXED VARIABLE>
           |{<OPERAND> | ( <EXPRESSION> )}
           <DYADIC OPERATOR><EXPRESSION>
<OPERAND>::=<VARIABLE>|<SCALAR VECTOR>|<CHARACTER VECTOR>
           |<UNSPECIFIED 0-ARG FUNCTION>|<INPUT SYMBOL>
<VARIABLE>::=<NAME>{<INDEXED VARIABLE>}^1
<INDEXED VARIABLE>::= [{<EXPRESSION>}^1{ ;{<EXPRESSION>}^1}]
<SCALAR VECTOR>::=<NUMBER>|<SCALAR VECTOR> <NUMBER>
<CHARACTER VECTOR>::= ¨ <CHARACTER STRING> ¨
<MONADIC OPERATOR>::=<BASIC MONADIC OPERATOR>
           |<DEFINED MONADIC OPERATOR>
           |<MIXED MONADIC OPERATOR>
<DYADIC OPERATOR>::=<BASIC DYADIC OPERATOR>
           |<DEFINED DYADIC OPERATOR>
           |<MIXED DYADIC OPERATOR>
<BASIC MONADIC OPERATOR>::= +|-|×|÷|*|~|⌈|⌊|||○|⍟|!|?|⍳|⍴
           |,|⍕|⌽|⍋|⍒|⍉
<BASIC DYADIC OPERATOR>::= +|-|×|÷|*|∨|∧|⍱|⍲|<|≤|=|≥|>|≠|⌈
           |⌊|○|⍟|!|?|||⍳|⍴|,|∊|↑|↓|⌽|⍕|/|⌿|\|⍀|←
<MIXED MONADIC OPERATOR>::={<BASIC DYADIC OPERATOR>
           |<DEFINED DYADIC OPERATOR>}{/ | ⌿}
<MIXED DYADIC OPERATOR>::={ ∘ |<BASIC DYADIC OPERATOR>
            |<DEFINED DYADIC OPERATOR>} .
            {<BASIC DYADIC OPERATOR>|<DEFINED DYADIC OPERATOR>}
<DEFINED MONADIC OPERATOR>::=<SPECIFIED 1-ARG FUNCTION>
<DEFINED DYADIC OPERATOR>::=<SPECIFIED 2-ARG FUNCTION>
<STATEMENT NUMBER>::={<DIGIT>}_1^4{.{<DIGIT>}_1^1}^1
<LABEL>::=<NAME>
<NAME>::=<LETTER>{<LETTER>|<DIGIT>}^3
<NUMBER>::= {¯}^1{<INTEGER FORM>|<DECIMAL FORM>
           |<EXPONENTIAL FORM>}
<INTEGER FORM>::={<DIGIT>}_1^6
<DECIMAL FORM>::={<DIGIT>}^i . {<DIGIT>}^j
<EXPONENTIAL FORM>::={<INTEGER FORM>|<DECIMAL FORM>}
           <EXPONENTIAL PART>
<EXPONENTIAL PART>::= E {¯}^1{<DIGIT>}_1^2
<DIGIT>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<LETTER>::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W
          |X|Y|Z
<OPERATOR SYMBOL>::= +|-|×|÷|*|∧|∨|⍲|⍱|~|<|≤|=|≥|>|≠|⌈|⌊|○
           |||⍟|!|?|⍳|⍴|,|∊|↑|↓|⍕|⌽|⍉|⍋|⍒|/|⌿|\|⍀|∘|←
<INPUT SYMBOL>::= ⎕ | ⍞
<SPECIAL SYMBOL>::= .|¯|'|→|:|;|(|)|[|]|¨|∆|⍝
<SYMBOL>::=<OPERATOR SYMBOL>|<INPUT SYMBOL>|<SPECIAL SYMBOL>
```

走査表 I （付録 II）

IPC / LP	:	←	→	(	)	[	]	;	関数 f	/ f	演算子 ⊕	.	□ ⊡	変数,定数 a	エンドマーク ε
S (S_N)	d_1: 名札処理 d_8 d_2: エラー	d_1: LS: pd, L_P(←), S→S_N d_8 d_2: エラー	d_1: エラー d_2: LS: pd, L_P(→), S→S_N d_8	d_1: エラー	エラー	d_1: LS: pd, L_P( ), BC←BC+1 d_8	エラー	d_1: LS: pd, L_P(;), S_N→S d_2 d_2: エラー	d_1: LS: pd, L_P(f_n) d_1	d_1: LS: pd, L_P(f_1) or L_P(f_2) d_8	d_1: LS: pd, L_P(⊕) d_8	エラー	d_1: エラー	d_1: エラー	d_1: if BC≠0 or PC≠0 エラー else LS: pd, L_P ε, 動作終了
・ 以外	エラー	d_1: LS: pd, L_P(←) d_8	エラー	d_8: LS: pd, L_P( ), PC←PC+1 d_8	d_1: LS: pd, L_P( ), PC←PC−1, if PC<0 エラー d_1	d_1: エラー	d_1: LS: pd, L_P( ), BC←BC−1, if BC<0 エラー d_1	d_1: if BC=0 LS: pd, L_P(;), S_N→S else LS: pd, L_P(;) d_2	d_8: LS: pd, L_P(f_n) d_1	d_1: LS: pd, if L_{P-1}=f_n, L_P(f_1) or L_P(f_2) else L_P(f_1) or L_P(f_1) d_1	d_1: LS: pd, if L_{P-1}=f_n, L_P(⊕), f_n→f_n else L_P(⊕) d_1	d_1: エラー d_2: LS: pd, L_P(・) d_2	d_1: LS: pd, L_P(□) or L_P(⊡) d_1	d_1: L_P(a) d_1	d_8: if L_P=→ if L_P=f_n f_n→f_n d_1 の場合と同じ操作 else エラー
・		d_2: エラー	d_1: エラー d_8: if L_{d-1}≠$\updownarrow$ LS: pd, L_P( ), PC←PC+1 d_2 else エラー	d_1: if L_P=f_n, f_n→f_1 d_1 の場合と同じ操作 else エラー	d_1: if L_P=f_n, f_n→f_n if L_P=[or L_P=; d_1 の場合と同じ操作 else エラー	d_1: if L_P=f_n, f_n→f_n if L_P=[d_1 の場合と同じ操作 else エラー	d_8: if L_P=f_n, f_n→f_n if L_P=[d_1 の場合と同じ操作 else エラー	d_8: if L_P=f_n, f_n→f_n if L_P=[d_1 の場合と同じ操作 else エラー	d_1: LS: pd, L_P(f_2) d_8 d_1: L_{d-1}(f_2) d_2	d_1: LS: pd, Lp(f_2) or L_P(f_2) d_1 d_8: エラー	d_1: LS: pd, L_P(⊡) d_8 d_8: L_{d-1}(⊡) d_1	エラー	d_1: エラー d_2: if L_{d-1}≠$\updownarrow$ LS: pd, L_P(□) or L_P(⊡) d_1 else エラー		

凡例：d_1：(dフラグが1ならば)，LS：pd(L-スタック・プッシュダウン)，L_P(←)(L-スタックP部に←のエントリ・ポイントを入れる)
PC←PC+1((格子のカウントを1つ増す)，L_{d-1}≠$\updownarrow$(Lスタックd_{d-1}部が空である)，S→S_N(SをS$_N$にかえる)

表査表 II （付録 III）

(R順位)		0	0	0	0	0	2	2	2	2	2	2
L順位	R_P / L_P	添字 ;	添字]	)	出力 ⊙•	エンドマーク ε	2項演算子 $\oplus_2$	縮小 $/_2$,f_2	2項関数 f_2	内積, 外積 ・	代入 $\leftarrow_v$,$\leftarrow_A$	配列 $[_{1V}$,$[_{2V}$
1	S	エラー			≫ステップ (出力, M1, R_d) RS: pop up	≫ステップ (出力, M2, R_d) (ε, M) ㊧ ≫ステップ: (ε, M) ㊨	∨ ステップ (R_P, R_d, R_{d-1}, TEMP) RS: pop up R_d←TEMP	∨ ステップ (R_P, R_d, R_{d-1}, TEMP) RS: pop up R_d←TEMP	∨ ステップ (R_P, R_d, R_{d-1}, TEMP) RS: pop up R_d←TEMP	∨  Rスタック if f≠0 内積 ステップ: (・, f, g, R_d R_{d-1}, TEMP) RS: pop up 2回 R_d←TEMP if f=0 外積 ステップ: (・, g, R_d, R_{d-1}, TEMP) RS: pop up 2回 R_d←TEMP	∨ if R_P=←$_v$ ステップ: (←$_v$, R_{d-1}, R_d) R_d←R_{d-1} RS: pop up if R_P=←$_A$ ステップ: (←$_A$, R_{d-1}, R_d) RS: pop up	∨  if R_{P-1}=←$_v$ R_P←←$_A$ ステップ: ($[_{1A}$, R_d, T, TEMP) if R_{P-1}=←$_v$ ステップ: ($[_{1A}$, R_d, T, TEMP) RS: pop up 2回 R_d←TEMP  if R_{P-1}=←$_v$ R_P←←$_A$ ステップ: ($[_{1A}$, R_d, T_1, T_2, TEMP) if R_{P-1}=←$_v$ ステップ: ($[_{1A}$, R_d, T_1, T_2, TEMP) RS: pop up 3回 R_d←TEMP
1	S_N											
1	⊙				≫ LR							
1	→					≫ステップ: (→, R_d) (ε, M) ㊨						
1	←	≫ LR										
1	$\oplus_2$	LR [RS: push down, L_P←R_P, L_{d-1}→R_d, LS: pop up]										
1	$/_2$, f_2											
1	f_2											
1	・	≫ RS: push down 2回, L_P→R_P, L_{P-1}→R_{d-1}, L_{d-1}→R_d, LS: pop up 2回										
1	$\oplus_1$	≫ ステップ: (L_P, R_d, TEMP) LS: pop up R_d←TEMP										
1	$/_1$, f_1	≫ ステップ: (L_{P-1}, L_P, R_d, TEMP) LS: pop up 2回, R_d←TEMP										
1	[	≫ L_P←$_{1V}$ LR	≫ L_P←$_{1V}$ LR	エラー								
1	;	エラー	≫ LR									
1	(			≫ R_d→R_{d-1} LS, RS: pop up								
2	f_{01}	≫ if R_d=$\updownarrow$ ステップ: (f_0, TEMP) LS: pop up, R_d←TEMP			if R_d≠$\updownarrow$ ステップ: (f_0, R_d, TEMP) LS: pop up, R_d←TEMP		± if R_d=$\updownarrow$ L_P≫R_P		if R_d≠$\updownarrow$ R_P≫L_P			
3	□, ⊡	≫ 入力要求 ステップ: (L_P, TEMP) LS: pop up, R_d←TEMP									≫ ステップ: (出力, M_3, R_{d-1}) LS, RS: pop up	≫ 入力要求
3	]	≫ LR										エラー
3	)										エラー	≫ LR

凡例　RS：push down (R-スタックをプッシュダウン)，LS：pop up (L-スタックをポップアップ)，L_P→R_P(L$_P$部をR$_P$部に移す)，R_d←TEMP(R_d部に1時記憶のエントリ・ポイントを格納する)
ステップ：(L_P, R_d, TEMP)，(ステップ例 L_P, R_d, TEMPを作る)，M, M_3, M_2, M_1はモディファイアである。