APL Interactive Processing System on a Minicomputer

Toyohide WATANABE*, Fuzio MIYAWAKI**, Katsumasa WATANABE***
and Hiroshi HAGIWARA***

Abstract

This paper deals with the presentation of an 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. First it scans from left to right and then from right to left. Moreover, during execution of all APL statements, dynamic storage allocation is used. Since there are insufficient data storage in the main memory, this system makes use of different kinds of data management techniques.

We will describe the processing method of the APL interpreter and show the design concept used to make APL match a minicomputer's characteristic.

1. Introduction

APL(A Programming Language)[1] was first defined by K.E.Iverson as a machine-independent programming language whose descriptive and analytic powers were adequate to consisely describe algorithms. There are features which distinguish APL from the more traditional programming language such as FORTRAN[2]. Some of its features include; (1) it allows a clear and simple representation of the sequence in which speps of the expression are evaluated; (2) it provides a concise and mathematical notation for the operations occurring in a wide range of processes; (3) it permits the description of a process to be independent of the choice of a particular representation of the data.

We were interested in actually making use of a programming language with these features and have implemented such a processing system on a minicomputer. From the point of view of the implementation, we aimed at making APL match a minicomputer's characteristic.

2. Outline of Implementation

Processing Method

This system takes advantage of an interpreter approach, with a consequent loss of execution speed. The source language is translated into an intermediate language which we call the step-sequences, and then the interpreter continuously evaluates it. Fig.1 shows the data flow on this system. Here the processing data divides the processing path into two modes: execution and definition. In execution mode, this system produces the intermediate language and then evaluates it; and in definition mode, this system only preserves it in system file without evaluation.

This paper first appeared in Japanese in Joho—Shori (Journal of the Information Processing Society of Japan), Vol. 16, No. 9 (1975), pp. 781~788.

* Data Processing Center, Kyoto University
** Himeji Institute of Technology
*** Department of Information Science, Kyoto University

With the absence of type declarations and DIMENSION statements, an attribute of the variables can't be decided until values are actually assigned to the variables during execution. Accordingly, we can't adopt the use of a compiler to generate efficient codes. But, when defined functions are called, this system evaluates the step-sequences preserved in definition mode because it is not very effective to evaluate the source pattern directly.

System Construction

The hardware construction consists of HITAC-10(12kW), Magnetic Drum(131kW) and a typewriter with an APL typing-element.

The number of the program steps are more than 20,000 words; 1 word is 16 bits. Also, this system deals with many areas and tables. Therefore, an overlay method is required. The memory map in Fig.2 shows two phases. Namely, as the processing procedure is composed of a translation part and an interpretation part, the overlay structure of this program is also constructed to make system performance more effective. We designed with a view in: (1) making system performance more effective; (2) making this system match a minicomputer's characteristic; (3) making the program structure clear.

3. Translation into Intermediate Language

Translation Process

The syntax of APL has characteristics as follows; (1) a statement is evaluated sequentially from right to left, but an expression in parentheses is looked on as an operand; (2) there are not priorities among many primitives; (3) many primitives operate not only as unary operators but also as binary ones; (4) there are neither type declarations nor DIMENSION statements; (5) whether or not a statement is an output one depends on its sentential form.

The analyzer has to scan from right to left because the evaluation is from right

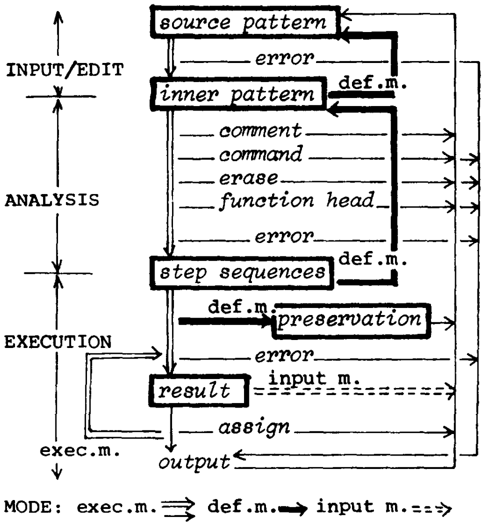


Fig.1 Flow of Processing Data

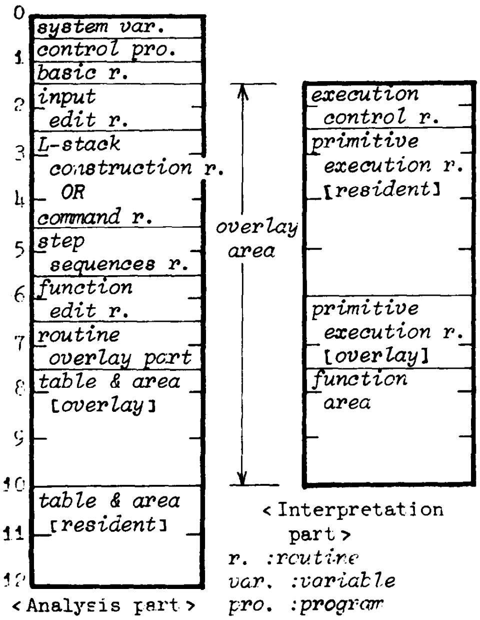


Fig.2 Memory Map

to left. But we adopt two-step scanning. First, when scanning from left to right, the analyzer edits variables and constants and distinguishes an unary operator from a binary one by whether there is a left operand or not. Next, it produces step-sequences on scanning from right to left. This is because it can't scan string letters continuously if a parsing process is only one step.

.Scanning from left to right

The analyzer translates an inner pattern into an entry pointer for each table and fills in L-stack; L-stack with 2 tracks consists of the operator and operand parts. It is shown in Fig. 3. SCANNING TABLE I(Table 1) indicates an action by comparing an inner pattern with an operator of L-stack.

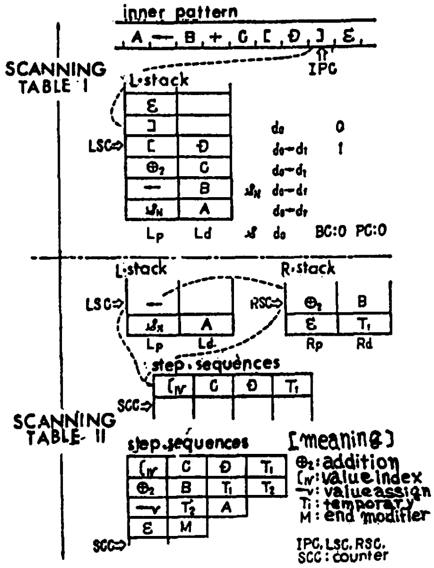


Fig. 3 Flow of Syntax Analysis from Innerpattern to Step-Sequences

In addition, this phase takes roles as follows: (1) distinction of either unary or binary operator; (2) decision of a sentential form; (3) level-count for "(" to ")", and "[" to "]".

The distinction between unary and binary operator depends on a D-flag(d_0,d_1 in Fig.3). This is a 2-state flag which points out whether the next primitive is either unary or binary; if the D-flag is d_0, the next one is unary; otherwise, binary. An action of SCANNING TABLE I starts out first by checking the D-flag and stops by re-setting up the D-flag.

.Scanning from right to left

In this step, the analyzer transforms the entry pointers into step-sequences. An action of the analyzer under SCANNING TABLE II(Table 2) is selected from a combination of the primitives between L-stack and R-stack. Basically, i). when $L_p^* > R_p^*$, (a) a step-sequence is composed of L_p.part in L-stack(in the case of unary primitive), or (b) parts of L-stack are transfered to R-stack(in the case of binary one).

ii). when $L_p^* < R_p^*$, a step-sequence is composed of R_p.part in R-stack.

iii). when $L_p^* \doteq R_p^*$, either i) or ii) is selected.

Remarkable Point at This Stage

A statement which contains an unde-

Table 1 **SCANNING TABLE I**

IPC \ LP	:	←	→	(	)	[	]	;	func. f	//	primi. ⊕	.	□ ⊡	var.; con. a	end mark ε
S (S_1)	d_1: label d_0	d_1: LS: pd L/(←) S→S_1 d_0	d_1: error	d_1: error	error	d_1: LS: pd L/([) BC←BC+1 d_0	error	d_1: LS: pd L/(;) S_1→S d_0	d_1: LS: pd L/(f_1) d_0	d_1: LS: pd L/(f_1) or L/(f_1) d_0	d_1: LS: pd L/(⊕) d_0	error	d_1: error	d_1: error	d_1: if BC≠0 or PC≠0 error else LS: pd L/(ε) end
	d_0: error	d_1: error	d_0: LS: pd L/(→) S→S_1 d_0					d_0: error							
except	error	d_1: LS: pd L/(←) d_0	error	d_0: LS: pd L/(() PC←PC+1 d_0	d_1: LS: pd L/()) PC←PC−1 if PC<0 error d_1	d_0: error	d_0: LS: pd L/(]) BC←BC−1 if BC<0 error d_1	d_1: if BC=0 LS: pd L/(;) S_1→S else LS: pd L/(;) d_0	d_0: LS: pd L/(f_0) d_0	d_0: LS: pd if L_{p-1}=f_0 L/(f_0) or L/(f_1) else L/(f_1) or L/(f_1) d_0	d_0: LS: pd if L_{p-1}=f_0 L/(⊕) f_0→f_1 else L/(⊕1) d_0	d_1: error d_0: LS: pd L/(·) d_0	d_0: LS: pd L/(□) or L/(⊡) d_1	d_0: L/(a) d_1	d_0: if L_p=→ if L_p=f_0 f_0→f_1 god1 else error –
		d_1: error		d_1: error d_0: if L_{p-1}≠φ LS: pd L/(() PC←PC+1 d_1 else error	d_0: if L_p=f_0 f_0→f_1 god1 else error		d_0: if L_p=f_0 f_0→f_1 if L_p=[or L_p=; god1 else error	d_0: if L_p=f_0 f_0→f_1 if L_p=[god1 else error	d_1: LS: pd L/(f_1) d_0 d_1: L_{p-1}(f_1) d_0	d_1: LS: pd L/(f_1) or L/(f_1) d_0 d_1: error	d_1: LS: pd L/(⊕1) d_0 d_1: L_{p-1}(⊕1) d_0	error	d_1: error d_0: if L_{p-1}≠φ LS: pd L/(□) or L/(⊡) d_1 else error		

tined function can't completely be analyzed in spite of the 2-step scanning. If a result of syntactical analysis is different from that of the definition, this system rearranges the step-sequences at the stage of the function call. This fact depends that with the absence of declarations it is not completely determined until a function is defined. But this problem could be avoided if such a system adopted a method such that on the function call inner patterns are translated into step-sequences. We adopted this method which preserves step-sequences in order to make evaluation performance more effective.

Consequently, this method contains other merits: it does not have to register a defined function uselessly by means of checking all of the statements of the function at an early stage. But, the demerits of this method are as follows:

i. on re-definition, one must use the same process to produce an intermediate language once again;

ii. it is difficult to reproduce a source language from an intermediate language;

iii. there is a problem where one can't exchange an attribute of the name freely.

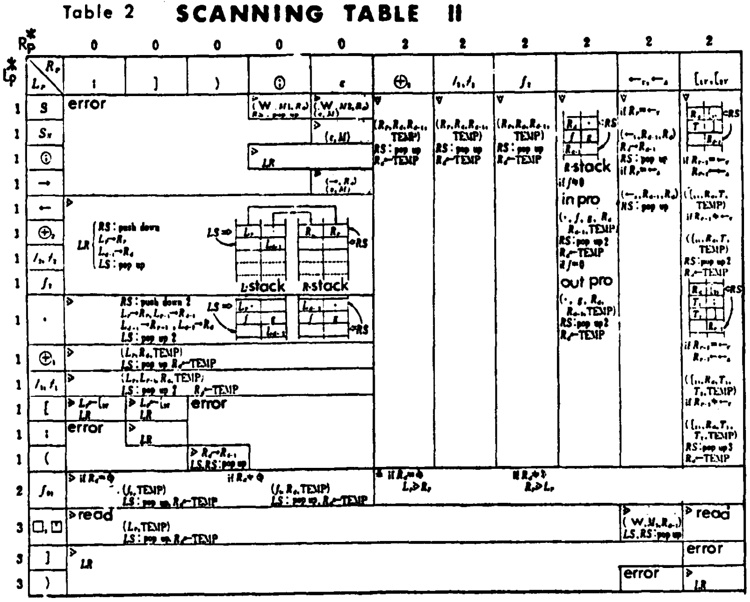
In the case of ii, we decided that the registered structure of the functions had both step-sequences and inner patterns; and in the case of iii, this system exchanges the attribute automatically only if the attribute transfers from variable to function.

4. Usage of Data Storage

Manipulation of Data Storage

As the attribute and size of all variables can change in an unpredictable way, the variables can not have fixed areas. For each evaluation step, a data area must be allocated for a variable. But, as there are no sufficient storage in the main memory, this system must have means to make use of limited storage as effective as possible. During execution, each primitive execution routine checks up the attribute of the operands and reserves a data area for the result. In this case,its routine checks up whether or not data areas which are allocated for temporary results can be reused, releases if possible and then allocates the areas. While, an end process routine releases areas which each routine couldn't do so; that is constant.

Table 2 **SCANNING TABLE II**



Structure of Data Storage

In addition to a useful manipulation of the data storage we divided it by itself into two kinds of areas: function data area and global data area. As constants used in defined functions are required only on evaluation, the system loads them to a fixed data area on the function call. This not only facilitates to edit constants of defined functions, but also gives merits to make use of the data

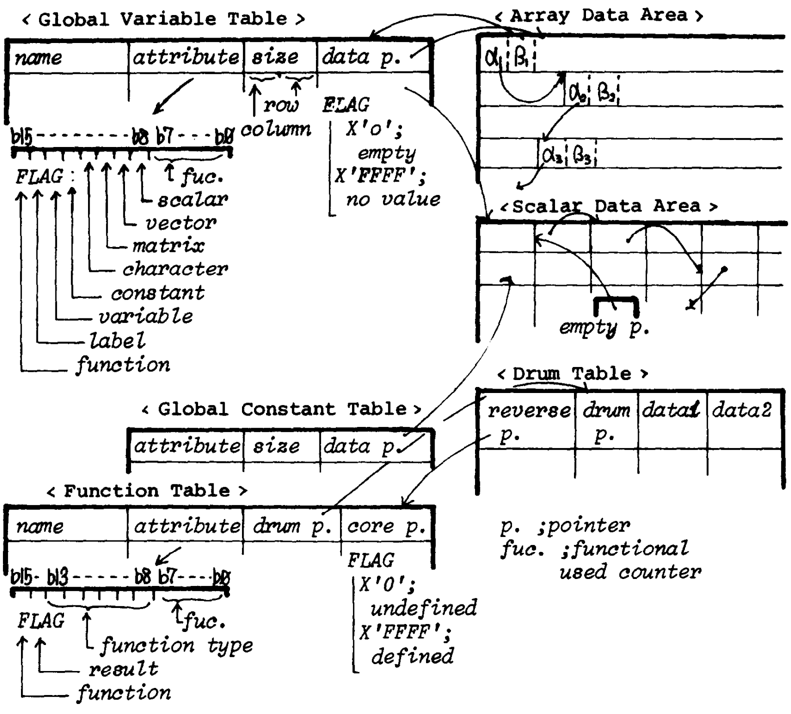


Fig.4 Data Entries

area. On the other hand, the global data area is futhermore divided into the scalar area and the array area. This is because we assume that the usage of a scalar is large even though APL is excellent enough to deal with arrays. In the scalar area, the data cell is 2 words; all empty cells are composed of a list structure. Also, the array area is a list structure of variable-length data cell which is attended with 2-word information cell. This information cell links each data cell in spite of whether or not each data cell is empty. One word is a pointer of the list structure (e.g. $\alpha_1,\alpha_2,\alpha_3$ in Fig.4), and the other one is a flag which indicates whether or not the cell is empty(e.g. β_1,β_2,β_3 in Fig.4). In the case of being empty, it points that the cell is empty(unused); otherwise, it is a reverse pointer to an entry table. A garbage collector first reconstructs the cells using reverse pointers. Second, it rewrites the data pointers. And finally it produces more data areas.

5. Conclusion

There are several problems we must consider. On the defined function, this system adopted the method such that the step-sequences were evaluated on the function call after preserved in the form of the intermediate language in order to make system performance more effective. This led to the problem of the transition of the attribute between functions and variables. But the problem must be investigated from the point of view of both effectiveness and dynamic features. In the future, new ideas will allow to make system performance effective[3,4].

REFERENCES

[1] Iverson,K.E. : A Programming Language, p.286, John Wiley & Sons.Inc., New York (1962).

[2] Falkoff,A.D. & K.E.Iverson : The Design of APL, IBM J.RES.DEVELOP., 17,pp.324-334(1973).

[3] Hassitt,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).

[4] Thurber,K.J. & J.W.Myrna : System Design of a Cellular APL Computer, IEEE trans. on computers, Vol.c-19, No.4, pp.291-303(April,1970).