CS 47

AN INTERPRETER FOR "IVERSON NOTATION"

BY

PHILIP S. ABRAMS

TECHNICAL REPORT CS47
AUGUST 17, 1966

COMPUTER SC IENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UN IVERS ITY

AN INTERPRETER FOR "IVERSON NOTATION"

PHILIP S. ABRAMS

Computer Science 239
Professor N. Wirth
Stanford University
May 30, 1966

CONTENTS

I. INTRODUCTION

Kenneth E. Iverson's book, A Programming Language [4], presented a highly elegant language for the description and analysis of algorithms. Although not widely acclaimed at first, "Iverson notation" (referred to as "the language" in this report) is coming to be recognized as an important tool by computer scientists and programmers.

The current report contains an up-to-date definition of a subset of the language, based on recent work by Iverson and his colleagues. Chapter III describes an interpreter for the language, written jointly by the author and Lawrence M. Breed of IBM. The remainder of the paper consists of critiques of the implementation and the language, with suggestions for improvement.

II. THE LANGUAGE

Although the Iverson language has previously been described in the literature [2,4,5,6,7], the subset implemented includes a number of hitherto unpublished changes and additions. This chapter will be devoted to a brief description of the current state of the language.

A. Data

Data are either scalars or arrays of scalars. A scalar is a numerical quanitity or a quoted alphanumeric character. Numbers are represented either as integers, terminated decimal fractions, or either of these two followed by a decimal scale factor, (e.g. 6.023~24, meaning 6.023×10^{24}) . A negative sign can be associated with a number or its scale factor, and is written above the line, as ¯36 or 1.6E¯19 . This is done to distinguish between the sign of a number and the operator '-' . A logical valued numerical datum has as its value either 0 or 1 .

Arrays can be of any rank. It is important to note the difference between multidimensional arrays and lists of lists. EULER [11,12], for example, represents a matrix as a list (ordered set) of lists, each of the latter representing a row. The disadvantage of this approach is that it singles out a particular coordinate, in this case rows, of the array, thereby making it difficult to deal with an arbitrary cross-section of an array. In this language, arrays have a symmetric structure, in the sense that no dimensions are distinguished. (This point is discussed further in Chapter V, A.)

The rank vector ρA of a datum A is a vector each of whose components is the dimension (number of elements) of the corresponding coordinate of A.

Hence, if A is a 5 by 7 matrix, we have $\rho A \equiv 5,7$ • ⊠ (Notationally, the elements of a vector are separated by commas.) If A is a scalar, then ρA is an empty vector, that is, a vector of no elements. Also, $\rho\rho A$ gives the rank or dimensionality of A . Hence, for a matrix, the rank is 2, while for a scalar, the rank would be 0 .

B. Identifiers

An identifier is defined in the usual way and can be used to name data or functions. In the former case, the identifier is called a variable. A value can be assigned to a variable by the statement

variable ← expression

The value of a variable is the most recent value assigned to that variable. The use of identifiers as function names is defined in section J of this chapter.

C. Indexing

It is often necessary to refer to a subpart of an array. This is done by the indexing operation. Indexing is indicated by a pair of square brackets containing a subscript list, immediately to the right of the quantity to be indexed, with the syntax shown below.

```
indable ::= varb
indable ::= const
indable ::= (expr)
slistl ::= sexp]
slistl ::= ]
slist ::= slistl
slist ::= ; slist
slist ::= sexp ; slist
indexedexp ::= indable[slist
```

*The sign '≡' will be used to denote mathematical equality, to avoid confusion with the operator '=' of the language.

The number of semicolons in the slist of an indable A must be $(\rho\rho A) - 1$. This subset of the language uses 1-origin indexing. Thus, for a datum A with rank vector ρA (and $(\rho\rho A) \geq 1$) the subscripts for the jth coordinate must fall in the range

$$1 \leq i_j \leq (\rho A)[j] \quad \text{for all } 1 \leq j \leq \rho\rho A$$

A subscript may be a vector, in which case a subarray is selected by the indexing operation. For vectors, define $A[i_1, i_2, \ldots, i_k]$ to mean $A[i_1], A[i_2], \ldots A[i_k]$. An analogous definition applies to vector subscripts on higher rank arrays. An empty subscript position is an elision meaning that the whole coordinate is to be selected. That is, if the ith subscript of A is elided, it is taken to be $\iota(\rho A)[i]$. Thus, for example, for a matrix M, M[I;] represents the Ith row of M. Indexing returns a result with the smallest possible rank. Thus, in the example just given, M[I;] is a vector.

D. Operators defined on scalars

1. The following simple binary operators (sops) are defined for scalar-valued numerical arguments. In all cases, the result is also scalar.

Operator	Function	Definition	Example
+	Addition	As usual	3+5 ≡ 8
-	Subtraction	" "	3-5 ≡ ¯2
×	Multiplication	" "	3×5 ≡ 15
÷	Division	" "	3÷5 ≡ .6
⌊	Minimum	A⌊B is A or B, whichever is smallest numerically	3 ⌊¯5 ≡ ¯5

(continued)

Operator	Function	Definition	Example
$\lceil$	Maximum	$A\lceil B$ is A or B, whichever is largest numerically	$3\lceil 5 \equiv 5$
$\mid$	Modulus	$R \leftarrow A \mid B$ is the least positive number such that for some integer Q, $B \equiv R + A \times Q$ $A \mid B$ is undefined when $A \equiv \infty$, and when both $A \equiv 0$ and $B < 0$.	$2 \mid 5 \equiv 1$
*	Exponentiation	As usual $A*B \equiv A^B$	$2*3 \equiv 8$
$\wedge$	Logical AND	" " } Arguments must be logical valued	$1 \wedge 0 \equiv 0$
$\vee$	Logical OR	" " }	$1 \vee 0 \equiv 1$
$<$ $\leq$ $=$ $\neq$ $\geq$ $>$	Relationals	$A \mathcal{R} B \equiv$ {1 iff $A \mathcal{R} B$ holds; 0 otherwise	$5 < 7 \equiv 1$ 'A' = 'T' $\equiv 0$
		(= and $\neq$ are defined in the same way for character scalars.)	

2. If A and B are not scalars but are arrays with identical rank vectors, then the operations defined above are applied to A and B element-by-element to produce a result with the same rank vector. For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} + \begin{bmatrix} 5 & 2 & 1 \\ -7 & 9 & -3 \\ -9 & 2 & -4 \end{bmatrix} \equiv \begin{bmatrix} 6 & 4 & 4 \\ -2 & 14 & 9 \\ -2 & 10 & 5 \end{bmatrix} .$$

If one argument of a sop is a scalar and the other is an array, the scalar is extended to match the other operand in rank. For example, this rule

gives $3 + (4,5,6,7,\bar{}9) \equiv 7,8,9,10,\bar{}6$. If any of the cases discussed so far holds, the two arguments are said to be compatible. A binary operation in which the operator is a sop is undefined if its arguments are not compatible.

3. The following unary operators (also sops) are defined on numerical scalars:

Operator	Function	Definition	Example
~	Complement	{ A must be logical valued ~A ≡ 1-A	~1 ≡ 0
+	Unary +	+A ≡ 0+A ≡ A	
	Unary -	-A ≡ 0-A	-5 ≡ ¯5
|	Absolute value	|A ≡ A⌈ -A	|¯5 ≡ 5
⌊	Floor	"greatest-integer-less-than" ⌊B ≡ B-1|B	⌊3.5 ≡ 3
⌈	Ceiling	"least-integer-greater-than" ⌈B ≡ - ⌊-B	⌈3.5 ≡ 4
*	Exponential	*A ≡ e*A, e ≡ 2.71828...	

If ⊙ is one of the unary operators above and A is non-scalar, then the value of ⊙A is the result of applying ⊙ to all elements of A .

4. There is no operator hierarchy for any of the sops or oops (Section F). Expressions are evaluated from right to left, with each operator, acting binarily if possible, using the first available operands. Parentheses may be used in the normal way to alter the order of evaluation of an expression.

E. Reduction

If A is a vector ($\rho\rho A \equiv 1$) then for any binary sop $\odot$, $\odot/A$ (read: $\odot$ reducing A) is defined as follows:

$$\text{If } Z \leftarrow \odot/A \text{ then}$$

$$Z \equiv A[1] \odot A[2] \odot \ldots \odot A[\rho A]$$

where precedence is from the right to the left, as usual. If A is empty, then Z is the identity element of $\odot$. For example,

$$\times/\iota 0 \equiv 1$$

If $(\rho\rho A) > 1$, then $\odot/A$ means reduction along the last coordinate of A. In general, reduction can be carried out over any coordinate of an array by subscripting the slash. We can define this general construction as follows:

$$\text{If } Z \leftarrow \odot/_j M \text{ then}$$

$$\rho Z \equiv (\rho M)[1], \ldots, (\rho M)[j-1], (\rho M)[j+1], \ldots, (\rho M)[\rho\rho M]$$

$$Z[i_1; i_2; \ldots; i_{j-1}; i_{j+1}; \ldots; i_{\rho\rho M}] \equiv$$

$$\odot/M[i_1; i_2; \ldots; i_{j-1}; i_{j+1}; \ldots; i_{\rho\rho M}]$$

for all combinations of the subscripts over the ranges

$$1 \leq i_K \leq (\rho M)[K], \text{ for all } K \equiv 1,2,\ldots, j-1, j+1, \ldots, \rho\rho M$$

From the above, if M is multidimensional, then

$$\odot/M \equiv \odot/_{\rho\rho M} M$$

F. Operators defined on non-scalars

There are a number of operators which are defined on non-scalars or whose results have a different rank vector from their argument(s). This class of operators, the oops (Other operators) is summarized below and defined in detail in section F.2. In the tables below, the following abbreviations are used:

o Any one-element quantity (A is a one element quantity iff $\times/\rho A \equiv 1$.)

v Vector

s Scalar

a Arbitrary rank

1. (a) Unary oops $\odot A$

Operator	Rank of arg	Rank of result	Function
ι	o	v	Interval vector
ρ	a	v	Rank vector
$\uparrow$	v	v	$1\uparrow A$
$\downarrow$	v	v	$1\downarrow A$
$\perp$	v	s	$2\perp A$

(b) Binary oops $A \odot B$

Operator	Rank of A	Rank of B	Rank of result	Function
α	o	o	v	Prefix vector
ω	o	o	v	Suffix vector
ι	v	v	v	Index vector
ϵ	v	v	v	Characteristic vector

Operator	Rank of A	Rank of B	Rank of result	Function
ρ	v	a	a	Replication
↑	o	v	v	Left rotation
↓	o	v	v	Right rotation
,	v	v	v	Catenation
⊥	v	v	s	Base value
⊤	v	o	v	Representation

2. Definitions of oops

(a) Unary oops

(i) Interval vector

$$\iota N \equiv \begin{cases} \text{undefined for nonintegral } N \text{ and } N < 0 \\ \text{empty vector if } N \equiv 0 \\ (\iota N - 1), N \text{ if } N > 0 \end{cases}$$

(ii) Rank vector

The operator ρ applied unarily to a datum A produces the rank vector of A . Its meaning is defined in section A.

(b) Binary oops

(i) Prefix vector

$$N \alpha J \equiv J > \iota N$$

(ii) Suffix vector

$$N \omega J \equiv (N - J + 1) \leq \iota N$$

(iii) Index vector

If $R \leftarrow X\iota Y$, then $\rho R \equiv \rho Y$

For $J \equiv 1,\ldots, \rho R$, $R[J]$ is the least K such that $X[K] \equiv Y[J]$. If no such K exists, then $R[J] \equiv 1 + \rho X$.

(iv) Characteristic vector

If $R \leftarrow M\epsilon C$, then $\rho R \equiv \rho M$ and

$R \equiv (\rho C) < C\iota M$

That is, $R[I] \equiv 1$ iff at least one component of C is equal to $M[I]$.

(v) Replication

If $A \leftarrow R\rho U$, then $\rho A \equiv R$

Let A' and U' be vectors formed from A and U respectively, by taking the components of each in row-major order. Then

$A'[J] \equiv U'[1 + (\rho U')\mid J - 1]$, for $J \equiv 1,2,\ldots, \times/R$

That is, A is built up in row-major order from the elements of U taken in row-major order, cycling on U as often as necessary. In particular, $0\rho A$, for any A, is an empty vector.

(vi) Rotation

Right rotation of U by J: $R \leftarrow J\downarrow U$

$R \equiv U[1 + (\rho U)\mid(\iota\rho U) - J + 1]$

Left rotation of U by J: $R \leftarrow J \uparrow U$

$R \equiv U[1 + (\rho U) \mid (\iota \rho U) + J - 1]$

(vii) Catenation

If $R \leftarrow A,B$ then R is a vector formed by appending the components of B to the right of A . That is,

$R \equiv A[1], \ldots, A[\rho A], B[1], \ldots, B[\rho B]$

(viii) Base value

If $R \leftarrow B \perp V$, with B and V compatible,

let $B' \equiv \begin{cases} B \text{ if } B \text{ is a vector} \\ (\rho V)\rho B \text{ if } B \text{ is a one-component quantity} \end{cases}$

and let W be defined as follows:

$W[\rho V] \equiv 1$

$W[I] \equiv W[I + 1] \times B'[I + 1]$,

for $I \equiv ((\rho V) - 1), \ldots, 1$

Then, $R \equiv +/W \times V$

(ix) Representation

$R \leftarrow V \top N$ then $\rho R \equiv \rho V$

R is a vector such that:

$V \perp R \equiv (\times/V) \mid N$ and $\wedge/V < R \equiv 1$

G. Generalized matrix product

The generalized matrix product, $M \odot_1 . \odot_2 N$, for $\odot_1$, $\odot_2$ any two binary sops, is a double operator between the operands M and N . M and N are compatible for matrix product if the dimensions of the last coordinate of M and the first of N agree. That is, if

$(\rho M)[\rho\rho M] \equiv (\rho N)[1]$.

If one of the operands is a scalar, it will be extended in the normal way to a vector matching the other operand.

In general, the result Z of the matrix product $M \odot_1 . \odot_2 N$ is defined as:

$$Z[i_1; i_2; \ldots; i_{(\rho\rho M)-1}; j_2; j_3; \ldots; j_{\rho\rho N}]$$

$$\equiv \odot_1/M[i_1; i_2; \ldots; i_{(\rho\rho M)-1};] \odot_2 N[; j_2; \ldots j_{\rho\rho N}]$$

for all values of the subscripts in the ranges:

$$1 \leq i_k \leq (\rho M)[k] \quad , k \equiv 1, \ldots, (\rho\rho M) - 1$$

$$1 \leq j_\ell \leq (\rho N)[\ell] \quad , \ell \equiv 2, \ldots, (\rho\rho N)$$

For example, suppose M and N are matrices. Then the familiar matrix product of linear algebra is given by: $Z \leftarrow M +.\times N$; for suppose that $\rho M \equiv m,n$ and $\rho N \equiv n,p$. Then from the above definition, we have

$$Z[I;J] \equiv +/M[I;] \times N[;J] \text{ for all } 1 \leq I \leq m$$

$$\text{and} \quad 1 \leq J \leq p$$

Other uses of the generalized matrix product are discussed in [4].

For two vectors X and Y, the <u>outer product</u> $R \leftarrow X \circ . \odot Y$, where $\circ$ is the null operator, is defined as:

$$\rho R \equiv (\rho X), \rho Y$$

$$R[I;J] \equiv X[I] \odot Y[J] \qquad \begin{cases} 1 \leq I \leq \rho X \\ 1 \leq J \leq \rho Y \end{cases}$$

As an example, $(\mathbb{M}) \circ . = \iota M$ is the identity matrix of order M .

H. Compression, expansion

The compression of a vector A by a compatible logical vector U is denoted by $X \leftarrow U/A$ and defined as follows:

$\rho X \equiv +/U$. Then X is derived from A by suppressing those elements A[I] for which $U[I] \equiv 0$. This operation can be defined by the program below.*

The expansion of a vector A by a logical vector U with $\rho A \equiv +/U$, is denoted by $X \leftarrow U\backslash A$ and has the following properties:

$\rho X \equiv \rho U$ and X is a vector such that

$U/X \equiv A$ and $(\sim U)/X \equiv (+/\sim U)\rho 0$.

These operations are generalized to arbitrary arrays in the same way as reduction.

```
    ∇X ← U/A
[1] I ← 1
[2] X ← 0ρA
[3] → (I > ρA)/0
[4] → (0 = U[I])/6
[5] X ← X,A[I]
[6] I ← I + 1
[7] →3
    ∇
```

I. Statements

The syntax for a statement (stmt) is

st ::= leftpart ← expression

* This program is written in the style described in section J.

```
st ::= → expression
st ::= expression
leftpart ::= varb
leftpart ::= varb[slist]
leftpart ::= □
stmt ::= st ↵
stmt ::= label : stmt
label ::= varb
```

Varb and varb[list] are to be interpreted as (possibly subscripted) variables. The symbol ' □ ' (box) suggests a blank page and denotes the output string. Assignment to □ causes the expression assigned to be evaluated and printed.

The symbol '→' (right arrow) designates a branch and is used to alter the flow of control in the execution of a function. Let $\mathcal{E}$ be the value of the expression to the right of the right arrow. If $\mathcal{E}$ is an integral single-component quantity and is within the range of the line numbers in the currently executed function, then control passes to the statement on line numbered $\mathcal{E}$; if $\mathcal{E}$ is an integral single-component quantity out of this range, the function is exited and control is passed to the point at which the function was entered. If $\mathcal{E}$ is an empty quantity, control is passed to the next statement if such exists; otherwise the function is exited as above. If none of these cases applies, the statement is undefined and (in an implementation) an error is indicated.

Each statement must begin on a new line, and the symbol '↵' in the syntax is an end-of-line marker indicating this. A statement consisting solely of an expression has as its effect the calculation of that expression. In general this effect is used to call a function.

J. Functions and programs

A function is defined by a program consisting of a head followed by a body of statements. The entire function definition is enclosed in

function quotes ' ∇ ' . The head establishes the function's name, the number of parameters, and whether or not it returns a value. A function definition has the following syntax:

```
functiondef ::= ∇ head ⍝ body ∇
head ::= headl
head ::= varb ← headl
headl ::= varb dfn varb
headl ::= dfn varb
headl ::= dfn
dfn ::= varb
body ::= stmt
body ::= stmt body
```

The varbs in the function head identify the parameters. If there is a left arrow preceded by a varb in the head, the function is expected to return a result. This is done by an assignment to this result variable within the body of the function. A function is invoked by mentioning its name in an expression, together with the appropriate number of parameters. A function has the same syntax within an expression as a binary or unary sop or a varb, depending on the number of parameters it takes. Actual parameters are transmitted to the function program by value.

Labels on statements in functions are varbs which are initialized to the line number on which they appear. These variables are non-local to the function and may be used in arithmetic expressions at will. Changing their values by assignment may affect their use as labels.

Example: The function below computes the GCD of two numbers.

```
∇X ← A GCD B
L1: X ← A
A ← A|B
B ← X
→ (A ≠ 0)/L1
∇
```

The function might be used later as follows:

$$R \leftarrow 4 + 6 \text{ GCD } 15 ,$$

in which case the value of R will be 7 at the completion of execution of this statement.

It should be noted that arguments to a function are passed exactly as they appear in the calling statement; that is, there is no extension as in the case of sops. Also, it is meaningless to use a binary function in a reduction or matrix product.

III. IMPLEMENTATION

The language defined in the previous chapter was implemented by an interpreter for the IBM 7090/7094 by the author and L. M. Breed of IBM. Except for a small number of machine dependent functions such as bit-pushing and type conversion, the entire system was written in FORTRAN IV to run under the IBSYS operating system. FORTRAN was chosen because it was the only high-level language available to both programmers.

This section describes the organization of the interpreter and discusses interesting techniques used in programming. For purposes of exposition, the organization of data and the logic of the program are described separately.

A. Data organization

All references to variables, constants, operators, defined functions, and temporary storage are made through a symbol table. In the interpreter, the symbol table is an array named S; its structure is shown schematically in figure 1.

All S-entries are either two or three machine words, depending on the class (syntactic category) of the entity represented. The first word is the class number (CLASS); the second is its base address (SPTR) in the M array; and the third, if present, is a pointer to a BCD print name in the high order part of S . The only entries which have print names are variables, function names, and language primitives.

In the program, the pointer to the symbol table entry under consideration at any time is generally in SYPTR, and CLASS and SPTR have values corresponding to S(SYPTR) and S(SYPTR+1), respectively. (In describing the program, FORTRAN notation will be used where appropriate.)

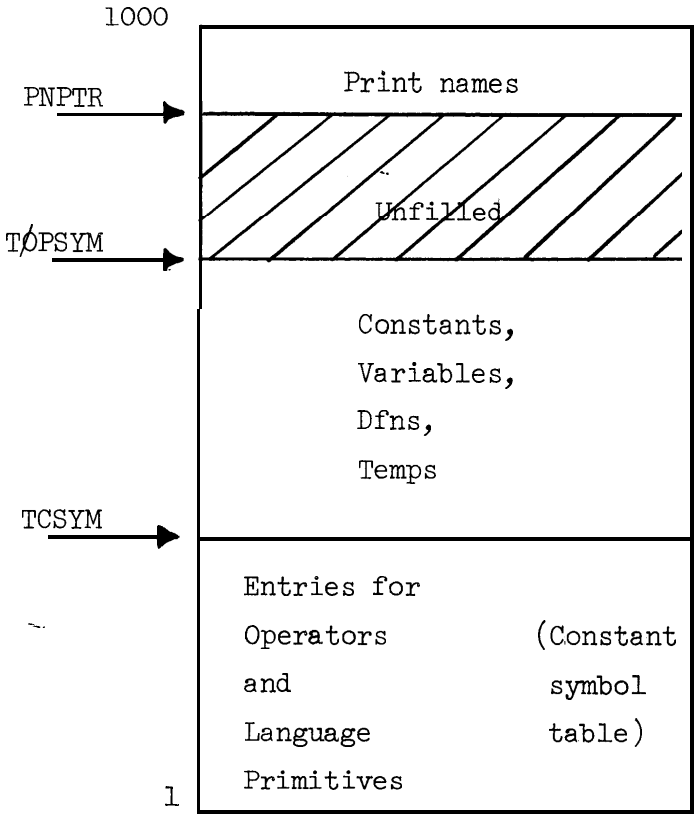


Figure 1: Symbol table (S) organization

The values of SPTR for operators and primitives are "magic numbers" which are used by the interpretation rules. For variables, constants, and temps, SPTR points into M to the value of the entity, and for function names (dfns) SPTR is the base address in M of the branch vector of the function.

TCSYM points to the top of the fixed symbol table; TOPSYM is the index of the next unused S location; PNPTR is the bottom of the print-name section. When TOPSYM > PNPTR, table overflow has occurred and an error is signalled. Note that storing the BCD print names from the top of S instead of in the lower part of S with the rest of the entry facilitates table searching, as there are no variable-sized entries in S.

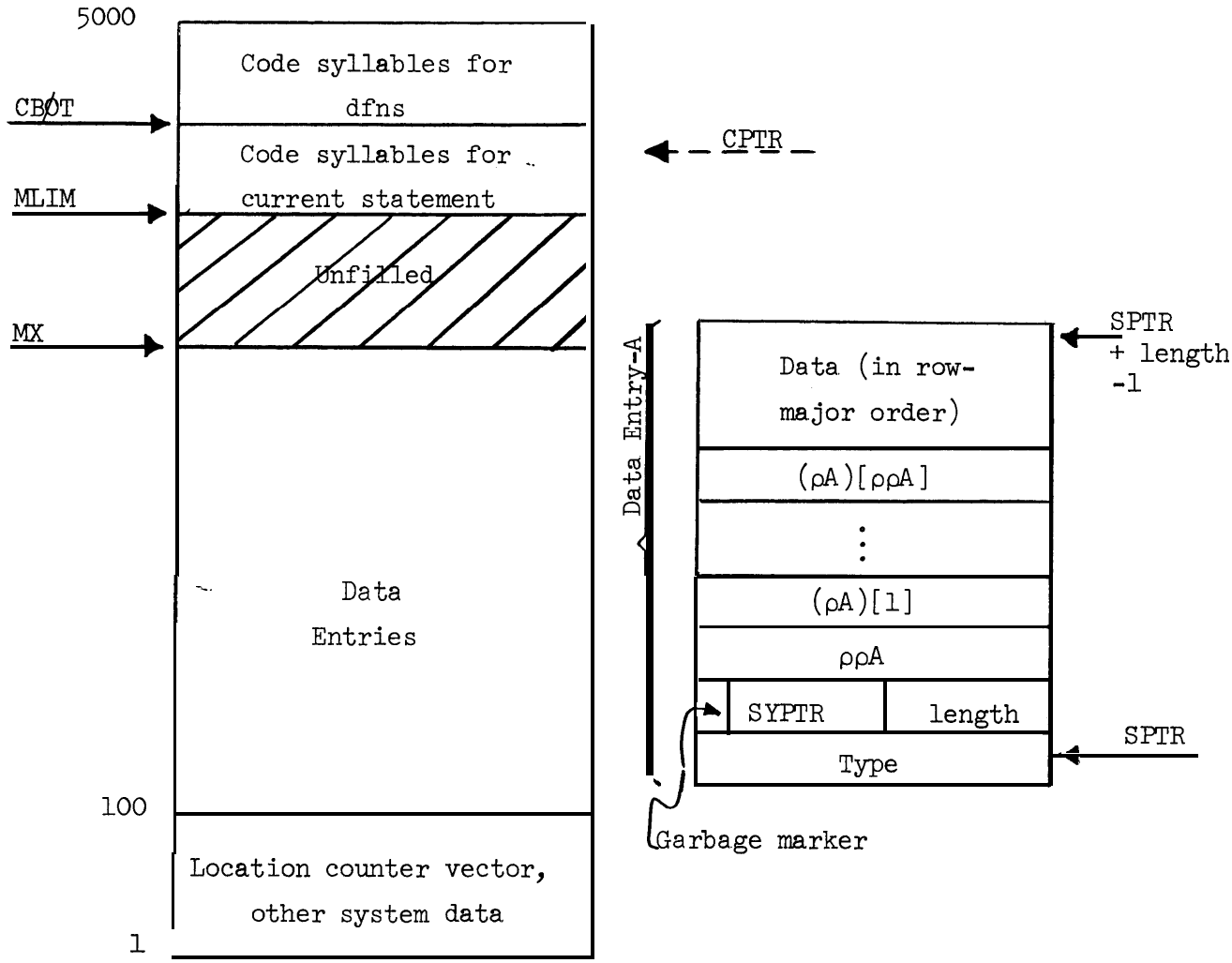


Figure 2: "Memory" (M) organization and typical data entry

The organization of M is shown in figure 2. Statements are translated into a code string of pointers to S and are stored down from the top of M . CBOT is the index of the bottom of the code for dfns, which is not changed, while MLIM is the bottom of the code for the current outer-level statement. MX points to the next piece of M available for data storage. All data space in M is allocated by the subroutine GETSPA. When N words are requested of GETSPA and MX + N $\geq$ MLIM, a garbage collection is made to reclaim abandoned space in M .

When an M entry with base address SPTR is no longer needed, it is marked as garbage by a bit in M(SPTR+1) . Garbage collection moves active entries down into abandoned space. The SYPTR and length entries are used by the garbage collector (GCOL) to update S for moved entries. One implication of this organization is that each M entry can be pointed to by only one S entry. This simplifies garbage collection but causes inefficiencies in M usage, as discussed in the next chapter.

Data in the system are classified into four type groups, which determine their internal representation. These types are:

1. Logical variables are represented as bit strings, packed 32 bits per machine word.
2. Integer values are represented as 36-bit sign-magnitude integers, 1 per word.
3. Floating values are represented in 7090 floating point format, 1 per word.
4. Character values are represented as 8-bit bytes, packed 4 characters per word.

For numeric values, quantities are represented as the lowest possible type in an attempt to conserve storage.

Each data entry in M contains the rank and rank vector of the data being stored. For multidimensional arrays M, the $\times/\rho M$ entries are stored in row-major order following the rank vector. That is, the mapping function used is exactly the base value function with the rank vector as radix. For example, if B is a floating array, the element $B[i_1; i_2; \ldots; i_{\rho\rho B}]$ has M index

$$SPTR + 3 + (\rho\rho B) + (\rho B)\perp(i_1, i_2, \ldots, i_{\rho\rho B}) - 1 .$$

B. The program

The purpose of the interpreter is to execute statements; that is, a statement is read, executed, and the cycle is repeated. In order to provide for programs with branching, such programs are defined as functions and are executed by calling the function.

Because of the limited character set of the 7090-1401 system in general use, it was necessary to transliterate the names of most of the symbols of the language. These are all reserved words in the system, and are part of the constant symbol table (see Figure 1). A table of correspondences between language symbols and their transliterations is given on the next page.

One of the two major subprograms in the system is TYPEIN, which scans each statement from left to right as it is read in and does the following tasks:

1. Recognizes reserved words and system symbols.
2. Creates symbol table entries for new identifiers.
3. Converts constants into M entries with matching S entry. In this process, constant vectors are treated as a single quantity, saving space in M and eliminating unnecessary catenation operations. For example, the statement

   X= Y MIN 3,4,7,9,46,Z - R DIV 3

   will be scanned and the underlined part will be entered into M as a 5-element vector rather than as five scalars and four operators.
4. Each statement is converted into a code string of pointers to appropriate entries in S, and these code syllables are

Symbol	Transliteration	Class	Meaning
+	+	16	Operators, see chapter II
		16	
×	*	16	
÷	DIV	16	
*	EXP	16	
⌊	MIN or FLOOR	16	
⌈	MAX or CEIL	16	
|	ABS or MOD	16	
∧	AND	16	
∨	OR	16	
<	LT	16	
≤	LE	16	
=	EQ	16	
≥	GE	16	
>	GT	16	
≠	NE	16	
~	NOT	16	
⍺	ALPHA	17	
⍵	OMEGA	17	
∈	EPS	17	
⍳	IOTA	17	
⍴	RHO	17	
↑	ROTL	17	
↓	ROTR	17	
⊥	BASE	17	
⊤	REP	17	
---	---	4	Temporary result (temp)
---	---	5	Variable (varb)
---	---	6	Constant (const)
[	$(	7	
(	(	8	
⌶	$)	9	
)	)	10	
;	.,	11	
:	..	12	
.	.	13	
←	=	14	
→	GOTO	15	
/	/	18	
	$/	19	
a	BOX	20	
o	NULL	21	Used in outer product
---	---	22	Function name (dfn)
---	---	23	Actual parameter (dummy)
---	---	24	Reserved word (used internally)
△	LOCN	25	Location counter
∇	DEFINE	--	Function quote
---	HYPHEN	--	Continuation to next card
---	DEBUG	--	Set diagnostic level
---	FINISH	--	End of run

Table 1: Language symbols and transliterations

stored in M from CBOT down. The left-most syllable of a statement is in the high part of M and the right-most has the lowest index in M. TYPEIN inserts a colon (:) as the left-most symbol in every code string, to be used by SYNTAX as a statement terminator.

5. When a function quote is encountered, TYPEIN sets an internal switch to change its mode from immediate execution to function definition mode. In this mode, the header of the function is scanned and the names of the formal parameters and the function are determined. As each statement is scanned, it is processed as described in steps 1 - 4 above. In addition, lines are numbered sequentially from 1, and when labels are encountered, they are given as value the current line number. The function name has as its value an integer vector of which the $i + 1^{st}$ element is a pointer to the right-most code syllable of statement (line number) i . The first element of this vector points to information obtained from the header, which is used for syntax checking.

   When an identifier corresponding to a formal parameter is scanned, a negative code syllable is emitted. These are interpreted as relative stack references by the syntax analysis, and are the mechanism for parameter linkage in function execution. Finally, when a closing function quote is found, TYPEIN returns to immediate execution mode, resets CBOT, and looks for the next statement.

6. When the end of an immediate statement is reached, TYPEIN terminates and control is passed to SYNTAX for statement execution.

The second major subprogram in the system is SYNTAX, which performs syntactic analysis of statements and controls execution through a series of interpretation rules.

Syntactic analysis is based on the separable transition diagram scheme of Conway [1]. In this scheme, the syntax of a nonterminal symbol of the grammar is represented by a transition diagram, the edges of which correspond to another grammatical symbol. To each edge there corresponds an interpretation rule (c.f. [11], [12]), which provides the semantics of the language. Each node in a diagram represents a set of alternatives. These are examined in a fixed order, thus providing a degree of context sensitivity. A circled edge from a node corresponds to "none of these" and is a default branch which is always satisfied if none of the others are. Self-recursion is replaced by looping within a given diagram. Figure 3 is an example of the diagram for stmt and the complete set of diagrams necessary to scan a statement is given in Appendix A.

The only syntax built into the transition diagrams is for a statement. Flow of control between statements is handled by the end-of-statement interpretation rule (S13 in figure 3). Also, note that in syntactic analysis, a statement is, in effect, scanned from right to left. Under the assumption that expressions will be written to take advantage of the right-to-left precedence rule of language operators, this scheme tends to conserve stack space.

The syntactic analysis described above is necessarily recursive; this recursion is handled by the "translator stack," ST (actually 2 arrays, ST1 and ST3 .) In SYNTAX, SI is always used as the stack pointer for ST .

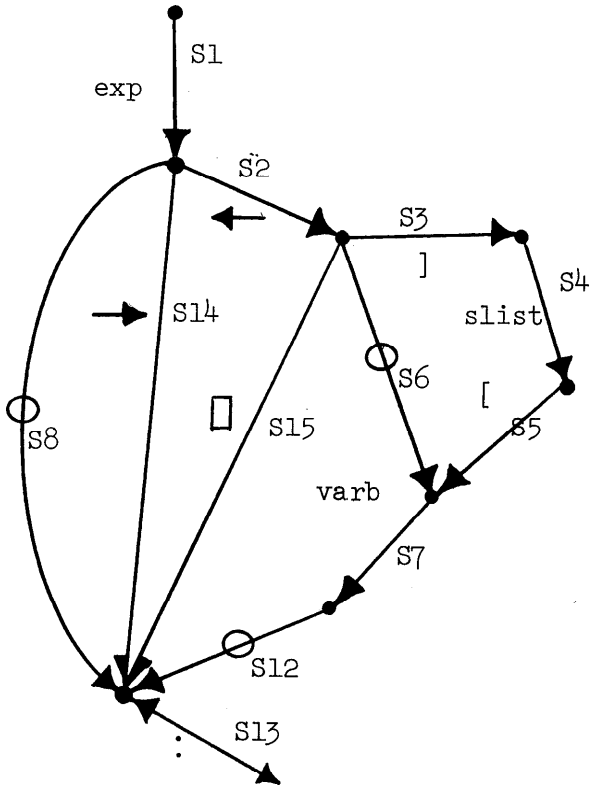


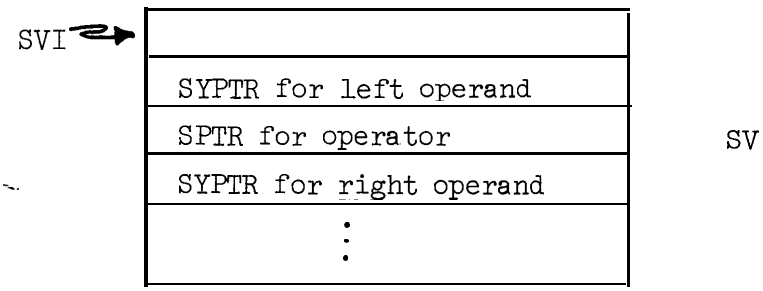
Figure 3: Transition diagram for stmt

There is also a "value stack," SV, indexed by SVI, which holds all temporary values (actually pointers to S) and function parameters. In general, each interpretation rule gets values from the top of SV, operates on them, and pushes the result(s) back into SV. The subprogram PUSH(V) puts V on the top of SV, increases SVI, and checks for stack overflow.

Interpretation rules, most of which are straightforward, make up the main body of SYNTAX. An examination of a typical rule will be instructive. The rule below is a simplified version of M6, which is encountered in traversing a smurg after having seen a sop followed by a basic.

```
980 SVI = SVI - 3
    T2 = SV(SVI)
    T1 = SV(SVI+2)
    CALL GETTEM(V)
    CALL DOSOP(V,T1,SV(SVI+1),T2)
    CALL PUTTEM(T1)
    CALL PUTTEM(T2)
    CALL PUSH(V)
    GO TO 205
```

When this rule is encountered, the stack looks like this:



The routine GETTEM(V) creates a symbol table entry of class temp (4) and assigns the SYPTR to V . DOSOP performs the operation coded in SV(SVI+1) on T1 and T2 and assigns the result to V . PUTTEM(T) marks temporary M storage for T as garbage and returns the symbol table entry for T to a linked temp list. This scheme keeps the number of temp S entries small. Thus, when GETTEM is called, it has to create a new S entry only if there are not any returned temp entries available. Finally, PUSH(V) pushes the result of executing the sop onto SV . The statement GO TO 205 returns control to the syntactic analyzer.

Within the system is a location counter vector, LOCN, which records the current line numbers of all active functions. In immediate execution mode, LOCN starts off as an empty vector. Each time a function is entered, a new element, starting at 1, is catenated to the right of LOCN.

When a function is exited, the last element of LOCN is deleted. The location vector is updated by function calls, the branch interpretation rule, and the end-of-statement interpretation rule. When a statement has been completely executed and LOCN is an empty vector, SYNTAX returns control to TYPEIN to read the next statement.

Function execution is straightforward. Actual parameters (if any) are copied to temp storage, if necessary, and are pushed into SV . If a result is indicated, a temp for the result is pushed. All actual parameters are given the type actual in S . Also on SV are put the SYPTR of the function name and certain global variables that record the current state of the syntactic analysis. A pointer FPTR is reset to indicate the innermost function being executed. LOCN is changed as described above, and execution of the function begun.

Upon exit, LOCN is reset, as are the global variables from the stack, and the result, if any, is pushed back into SV .

A complete list of the subroutines in the interpreter and their functions is given in Appendix B.

Extensive error checking is done in all parts of the program. When an error is detected, execution of the statement is abandoned and control is returned to TYPEIN to read and attempt a new statement. A diagnostic message indicates the cause of the error and the state of the interpreter when it was detected.

The interpreter includes almost all of the language described in chapter II. Those features which were not implemented are outlined below:

1. Subscripting of the operators / and \ is not in the system. The most obvious modifications to the transition

diagrams to allow this, also introduce syntactic ambiguities. When this feature is added to this or another interpreter, one would also like to add subscripting of some of the oops (for example $\uparrow, \downarrow, \perp$) to provide for their extension to multidimensional data. Such generalizations to matrices are descirbed in [4].

2. The mask and mesh operators [4] have not yet been programmed. (See chapter V, D)

3. In [4], compression on the left of an assignment arrow is allowed, as in the statement: $U/X \leftarrow 2,3,9$. Although this is a convenience, the same thing can be said using ordinary indexing: $X[U/\iota\rho X] \leftarrow 2,3,9$, and thus this feature was not included in the interpreter.

4. The constant high minus sign and the exponential form of constants have not been implemented.

IV. CRITIQUE OF PROGRAM

The interpreter just described has been thoroughly tested on a number of programs and appears to be reasonably bug-free. Little effort was put into any attempt to make the program efficient with respect to timing, and it appears that the interpreter is indeed rather slow. It is difficult to give meaningful timing figures, since each different kind of operator takes a varying amount of time; as an example of this, note the timings of the sample programs in Appendix C.

If the system were to be rewritten, there are several changes that should be considered, based on experience gained from this implementation. Some of these proposals have been suggested by L. M. Breed, based on work with the TSM system (see below).

1. The 7090 system is difficult to use because of the transliteration of symbols necessary to present a program to the machine. This problem can be solved by using an input device such as a CRT terminal or an IBM 1050 or 2741 typewriter terminal for which a typing element (type-ball) with the Iverson character set is available. For example, Breed adapted an earlier version of the interpreter for use on the now defunct TSM time-sharing system at IBM. With the inclusion of simple text-editing statements in the language, its usability was increased manyfold by being available at an online terminal with the proper character set.
2. Organization and allocation of M storage can be changed to simplify the interpreter and increase M usage efficiency.

There are several aspects to consider:

(a) The major reason for storing statement text as a series of S pointers was to allow for text editing and reconstruction of statements for error diagnostics. Editing does not exist in the current implementation but would be necessary in any online use of this system. Under the current arrangement, a special garbage collector would be needed to reclaim abandoned code space at the top of M . It thus makes sense to store the code string for a statement directly in M as, say, an integer vector. Then, the regular garbage collector can be used to reclaim abandoned text. This proposal will complicate the garbage collector, as there would be M entries, namely the branch vectors for the function names, which point to other (moveable) M entries; this problem is not very significant, however, since the addition of lists of the language (see Chapter V, A) requires an identical extension of the garbage collector. It would still make sense to put code strings for immediate execution statements into high M to eliminate the necessity of reclaiming the space thus used.

(b) It was found by users of the TSM system that in long work sessions, many constants were introduced in immediate execution statements which were no longer needed when these statements were completed. The net

result was that both S and M became filled with unused entries which were not reclaimable because there was no mechanism for marking them as garbage. A possible solution is to put a constant directly into the code string, preceded by a special syllable which marks the next entry as a constant. This would slightly complicate the problem of getting the next code syllable in the syntactic analysis, but would eliminate all constant entries in S, as well as left-over constants in M from immediate execution statements.

(c) Most of the M space marked as garbage is from abandoned temporary storage. In an earlier version of the interpreter, temp storage was stacked down from the bottom of the code string, and abandoned by changing MLIM when a statement was finished. This was unsatisfactory for two reasons: MLIM had to be stacked on SV whenever a function was entered; also, in a long statement using many temps, if M became full, a special garbage collector was needed to compact the temp storage abandoned but not yet erased.

One possible solution is as follows: In the execution of almost all of the sops and most of the oops, at least one of the operands is the same size as the result. Further, the execution of these operators is sequential. Thus, it should be possible to rewrite the operator execution programs for sops (DOSOP) and the appropriate oops (such as $\uparrow, \downarrow, \top, \epsilon, \iota$) to put the

result directly into the space occupied by the longer operand, if the latter is itself a temp.

3. The present method of syntactic analysis appears to be more powerful than necessary to treat this language. Even so, it is extremely simple to implement and is relatively compact. (The entire syntax analyzer is written in about one page of FORTRAN, and the diagram tables take less than 200 words of 7090 storage. This latter figure can be cut by at least a factor of 3 by judicious packing of the table.) One might still desire a simpler analysis routine, and at least two candidates for this position come to mind.

   (a) Rewriting the syntax so that it is a precedence grammar allows an even simpler analysis routine [11], [12]. However, a disadvantage is that in order to provide for complete error detection and recovery, the whole precedence matrix has to be kept in the program. In addition, the table of productions necessary for syntactic reduction would probably be at least as long as the present tables. The interpretation rules would probably be no more complex than those in the current scheme.

   (b) Another scheme which requires very little table storage and an extremely short analysis routine is as follows: A current state (essentially an indication of what is on top of the stack) is kept and compared to the syntactic class of the incoming symbol. If this

state-class pair is allowable, then an appropriate interpretation rule is invoked and state is altered; if not, an error is signalled. The simplicity of the scheme follows from the observation that a very small number of states and classes is necessary to define the syntax of the language. Thus, a short table of bits is sufficient to contain all the requisite information for the analyzer. A slight disadvantage is that the interpretation rules will probably have to be a little more complicated than at present in order to do extended error checking and operator execution.

With these two proposals in mind, it still appears that the transition diagram approach is most satisfactory for this and future interpreters. The primary reason for this is that using the diagram formulation, it is easier to alter the syntax of the language than in either of the other two schemes; this is particularly important in an experimental interpreter. Also, with the syntax represented in diagrams, much of the recursion which would normally occur in parsing can be replaced by iteration, which tends to conserve stack space.

V. CRITIQUE OF THE LANGUAGE

While I am a strong supporter of the Iverson language, I believe there are a number of areas where it is weak and could bear improvement. Almost all of these are points of omission rather than objections to features already in the language. This chapter is devoted to an outline of desirable new features, and should be considered as a set of suggestions for future work rather than detailed proposals.

The problem of adding new features to this language is not a trivial one. As it stands, the language is a powerful notation for describing processes, and is rich in formal identities. Any changes to the language should be consistent with the established body, both syntactically and in spirit. The danger of making ad hoc additions is ever present, and much thought will be necessary to work out the details of the suggestions that follow to avoid destroying the language or cluttering it with questionable kludgery.

A. Lists

The language currently has no provision for list-like structures. In his book [4], Iverson developed a subset of the notation to deal with trees. While powerful, it was wholly analytic; in order to construct a tree, one had to resort to building up a different representation of it, such as a right- or left-list matrix. Rather than extend this tree notation, I suggest a more "conventional" approach, along the lines used in EULER [11], [12].

Define a <u>list</u> to be an ordered set of elements, each of which can be a scalar, an array, or a list. Notationally, a list will be

represented as

$$\{\mathcal{E}_1; \mathcal{E}_2; \ldots; \mathcal{E}_n\}$$

where each of the $\mathcal{E}_i$ is a list element and the curly brackets are called list brackets. The use of the semicolon as a separator is consistent with the existing notation, in which subscript elements are separated by semicolons. Thus with lists, it becomes apparent that the construction $A[\mathcal{L}]$ where $\mathcal{L}$ is an slist, is really an abbreviation for $A[\{\mathcal{L}\}]$.

In adding lists, the available data space is made richer because lists extend it to include cartesian products of arbitrary subspaces, in the sense of McCarthy [8]. It is not desirable, however, to eliminate arrays as they exist in the language. A formulation of an array in list terminology makes it a list of lists of...of lists of elements. For example, a matrix becomes a list of rows (columns). The disadvantage of this approach is that it distinguishes some coordinates of an array over others, which for many purposes is undesirable. In different terms, considering arrays as lists of lists is to confuse the idea of an array, a purely mathematical concept, with its representation. In making the generalization to lists while retaining arrays it is tempting to consider the possibility of arrays of lists, but this, I think, is carrying a good thing a bit too far.

Given lists, it is necessary to define operations upon them. I propose the following as a start:

1. Catenation (appending) -- For A and B both lists, A,B is a list composed of catenating A and B at the top level. For example,

   {a ; b ; c}, {d ; e} ≡ {a ; b ; c ; d ; e}

The symbol ';' cannot be used for the list catenation operator because this would cause a conflict in the meaning of the symbol. For example, it would then be difficult to explain how {{1} ; {2}} represents a list of two elements, each of which is a list, as opposed to being a list whose sole element is the catenation of the lists {1} and {2} .

2. Arithmetic operations -- Arithmetic operations can be extended to compatible lists element-by-element, as is currently done for arrays. Here the definition of compatibility would have to require both identical structure and that the primitive elements at the lowest levels are numerical quantities which are array compatible.

3. Indexing -- A list can be indexed in order to select individual elements. If a subscript is a list of more than one element, then it will be interpreted to mean level-by-level selection. For example, this rule would give

$$\{1\ ;\{2\ ;3\ ;4\}\ ;\{5\ ;\{6\}\}\}[2\ ;\ 1] \equiv 2$$

Here, as in array subscripting, the use of square brackets around a list is actually an elision of an inner pair of list brackets. Using this convention for square brackets, there is no reason not to allow a list-valued expression to appear as a subscript within square brackets.

It would probably be desirable to allow vectors as subscripts to lists. However, I can think of no definition which would have the following property analogous to vector subscripting

of vectors:

$$L[\iota(\text{length}(L))] \equiv L$$

I submit that for the sake of consistency, we would like that property to hold, and that any definition should conform to it.

4. Structural operators -- In the absence of declarations, it should be possible to determine whether a datum is a list, as well as some information about its structure. Since lists as we have defined them are in some sense isomorphic to a generalization of Iverson's trees, one possibility for determining structure would be to use his analytic tree operators for moment vector, dispersion vector, number of leaves, and degree $\underset{\sim}{\mu}$, $\underset{\sim}{\nu}$, λ, δ, respectively.

All that is really necessary to use lists is a list predicate and a length function. The other functions mentioned above can be defined in terms of functions in the language. The predicate can be similar to the operator isli of EULER, and the unary ρ operator of the language can be interpreted to mean the length of the top level of a list when given a list as argument.

It should be possible to convert a vector to a list of its elements by a primitive operator. A suggestive notation for this is list $\leftarrow$; / vector, with the obvious definition. It might also be possible to extend the definition of the binary ρ operator to the construction of lists, but I have no clear notion of how this could be done.

B. Program structure

One of the most important features of the language is its ability to express, easily and naturally, operations on structured data. At the expression level, this is highly elegant. The structure of programs, however, is still at the level of machine language. When arrows along the side of the page are used to indicate branching (as in [4]), the structure of a program is equivalent to a flow chart, and is easy to follow. It is, however, inconvenient and often verbose to have to write all this flow information with explicit branches, as is necessary when a program is presented linearly to a computer.

I believe that a good programming language should make it possible to state an algorithm simply, in such a way that the complexity of the program corresponds in some straightforward way with the complexity of the algorithm it expresses. The current language has this property to a large degree, and the suggestions in this section are directed towards improving it in this area.

1. Iteration control -- The DO statement of FORTRAN and, even more so, the <u>for</u> statement of ALGOL 60 have proved to be very powerful and convenient mechanisms for iteration control. With the inclusion of lists in the language, a generalization of the <u>for</u> statement can be added quite easily.

   Let us allow the following construction:

   <u>for</u> X ϵ L do S

   where X is a variable, L is a list, and S is a statement. This statement is executed by letting X take on as

value successive elements of L, with S being executed for each such value. Also of value would be an optional while clause, as in ALGOL, and the statement

while R do S ,

for R any logical valued expression. This statement would evaluate R, execute S if the value of R is 1, and repeat the cycle as long as the value of R remains 1 .

One problem that appears immediately is that the proposed constructions allow only a single statement in the scope of an iteration. At least three ways of indicating scope come to mind: compound statements as in ALGOL; labeling the last statement in the scope as in FORTRAN; and indicating the number of statements in the scope. Of these three, I prefer the first as being the cleanest and most straightforward. As a convenience in writing compound statements and programs in general, it would be helpful to introduce an (optional) statement termination symbol, analogous to the ';' in ALGOL or PL/I, which allows several statements to be written on a single line.

2. Case analysis -- Almost all but the most trivial programs employ some form of case analysis; that is, execution of different parts of a program depending on some condition. In McCarthy's formalism [8], ALGOL 60, EULER, PL/I, and a proposed extension of ALGOL [13], among others, case statements, conditional statements, and conditional expressions have been provided to make this easily expressable.

At the program level, the case statement corresponds to indexing a pseudo-array of statements and as such, is a generalization of the conditional statement. Such a construct in the language would considerably shorten programs in the notation with no sacrifice in clarity. As an example of the usefulness of a case-type statement, consider the machine simulation example in Appendix C.

The need for a conditional expression or a generalization thereof is just as great in that it allows conciseness in expressions. It remains true that none of the constructions in this section add "power" to the language, in the sense that new things can be said which couldn't be said before; however, the goal of ease of expression suggests their necessity.

A re-interpretation of an existing construction can provide a generalization of the conditional expression analgous to the case statement. Given the expression

$$\{\ell_1 ; \ell_2 ; \ldots ; \ell_n\}[i]$$

let this mean selection of the ith element of the list, without evaluating the rest of the list.
For example, the factorial function can now be defined in a single statement (compare the same function defined using branching in Appendix C):

```
∇X ← FACT N
X ← {1 ; N × FACT N - 1}[1 + N ≠ 0]
∇
```

C. Functions

As currently formulated, functions (procedures) may have at most two parameters. Further, there is no mechanism for local variables within a function, which makes recursive definitions difficult. A proposed solution has been put forth by the Iverson group and is described below. Other questions to be considered are name parameters, functional arguments, and block structure.

Let the function header line be of the following form:

$$\nabla X \leftarrow F\ P_1 ; P_2 ; \ldots ; P_n$$

In using the function, the right hand parameter can be a list. When the function program is entered, the P_i are initialized to the corresponding list elements. If n is greater than the length of the list used as a parameter, the remaining P_i are undefined until values are assigned to them by the program. This scheme appears to solve both the problem of number of parameters and that of local variables rather handily. By introducing a function (operator) isdef such that isdef X is 1 iff X is defined (has a value) and 0 otherwise, it becomes possible to determine which of the P_i were initialized on a particular call of the function.

Some mechanism should be available to allow the use of name parameters, in the ALGOL 60 sense. I have no good ideas on how this could be fitted into the current notation. A similar situation holds for functional arguments to functions. Here, perhaps something on the order of McCarthy's use of λ-expressions would be workable, possibly using part of the available notation for function definitions. Implementationally, functional

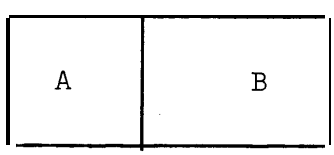
arguments open a Pandora's box of problems associated with variable bindings, so much thought will be required on this point.

The equivalent of block structure can easily be introduced given the mechanism for local variables discussed above. It is only necessary to allow function definitions to be nested to achieve this effect. This eliminates the need for an explicit block syntax.

D. Operators

As the notation is replete with powerful primitive operators, it is difficult to think of new ones which need to be added. The only situations in which this is justifiable are for functions which either are not definable in the notation, such as catenation, or which are sufficiently primitive and useful, yet complicated to program as defined functions. I will list the few primitives I think should be considered and give a few general remarks about each.

1. Ravel -- This would be a generalization or row- or column-list expansion of an array [4], and would decompose a higher dimensional array into a vector in an order specified by other parameters to the operator.
2. Laminate -- A generalization of catenation which juxtaposes two compatible arrays in a parametrically specified way. For example, if A and B are matrices with the same number of rows, then the lamination operation should be able to adjoin A to B as illustrated schematically below:



3. Transpose -- It is desirable to be able to obtain only only the regular transpose of a matrix, but to be able to permute the elements of any array in a number of specified ways. The Iverson group at IBM is working on a generalization of this operator.

4. Mask and mesh -- These two operators, part of the "classical" notation, while very elegant, are not generally useful enough to justify their being primitives in the language. A major use of mask (as, for example in [2]) has been as a special case of conditional expressions. With the adoption of the suggestions in Section B, the mask is no longer necessary in this context.

5. Set operators -- Using vectors to represent ordered sets, Iversion introduced set operators in [4]. With lists in the language, it seems more natural to let lists represent sets and to redefine these operators. On the other hand, if sets and set operations are sufficiently useful in a programming language, it may be more reasonable to introduce a new data type, the set. Such sets would be unordered and the operators defined on them could be introduced in such a way that they obey the laws of set theory for finite sets.

E. Independent programs

In machine descriptions (for example [2]) the use of independent programs (system programs) is necessary. There are no syntactic problems in allowing several independent programs, but many difficulties are imposed on an implementation. That is, as soon as system programs are

allowed in an interpreter for the language, all of the problems associated with simulation come to the fore. Ultimately, one would like to be able to execute several programs simultaneously, but the implementation of a system to allow this will be a major project in itself.

ACKNOWLEDGMENTS

The interpreter described in this paper was written jointly with Lawrence M. Breed of IBM Research. Without his effort and the countless conversations and arguments we had together, this work would not have been possible. I am especially grateful to Kenneth E. Iverson, who developed the language in the first place, and Adin D. Falkoff, both of IBM, for numerous discussions on the language and its philosophy. I also wish to thank John Lawrence of SRA for his encouragement and support; Michael Montalbano of IBM and Stanford for his help and enthusiasm; and my adviser, Professor Niklaus Wirth, for his many helpful criticisms and suggestions.

Parts of this work have been supported by the Computer Science Department of Stanford University, Science Research Associates, Inc., International Business Machines Corporation, and the National Science Foundation (Grant GP-4053).

PSA

REFERENCES

1. Conway, Melvin E., "Design of a Separable Transition-Diagram Compiler," Comm. A.C.M., 6, 7 (1963) 396-408.

2. Falkoff, A.D., K.E. Iverson, and E.H. Sussenguth, "A Formal Description of SYSTEM/360," IBM Systems J., 3, 3 (1964) 198-262.

3. Hellerman, H., "Experimental Personalized Array Translator System," Comm. A.C.M., 7, 7 (1964) 433-438.

4. Iverson, Kenneth E., A Programming Language, Wiley, New York (1962).

5. -------, Elementary Functions, Science Research Associates, Chicago (1966) In Press.

6. -------, "Formalism in Programming Languages," Comm. A.C.M., 7, 2 (1964) 80-88.

7. -------, "Programming Notation in Systems Design," IBM Systems J., 2, 2 (1963) 117-128.

8. McCarthy, John, "A Basis for a Mathematical Theory of Computation," in Computer Programming and Formal Systems, North-Holland Publishing Company, Amsterdam (1963) 33-70.

9. ------et al., LISP 1.5 Programmer's Manual, M.I.T. Press, Cambridge (1962).

10. Naur, Peter (ed.), "Revised Report on the Algorithmic Language ALGOL 60," Comm. A.C.M., 6, 1 (1963) 1-17.

11. Wirth, Niklaus and Helmut Weber, EULER: A Generalization of ALGOL, and its Formal Definition, Computer Science Dept., Stanford Univ., Technical Report CS20 (April 27, 1965).

12. ------, and ------, "EULER: A Generalization of ALGOL, and its Formal Definition" Comm. A.C.M., 9, 1&2 (1966) 13-23 and 89-99.

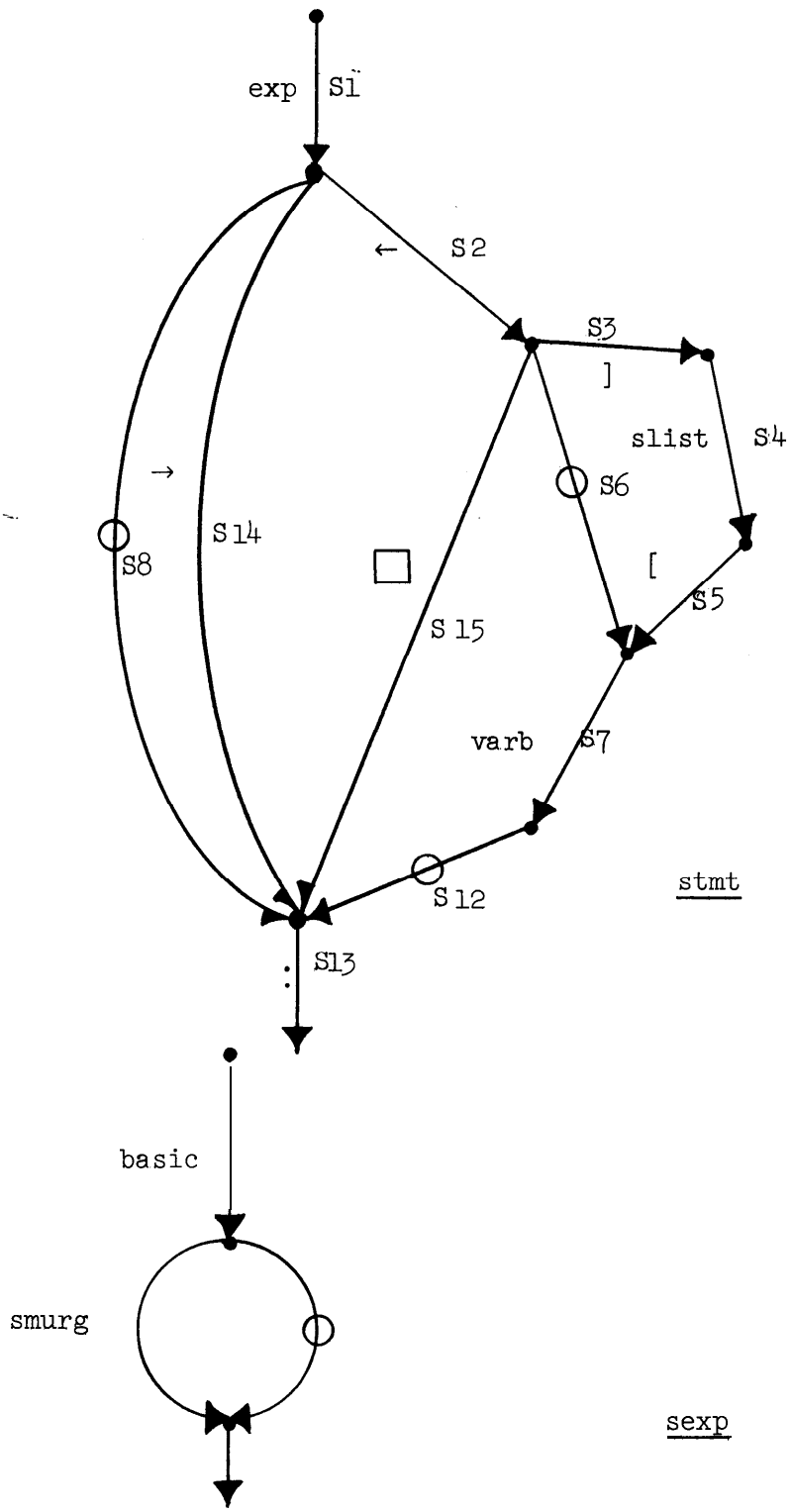
13. ------, and C.A.R. Hoare, A Contribution to the Development of ALGOL, Computer Science Dept., Stanford Univ., Technical Report CS35 (February 12, 1966, revised).

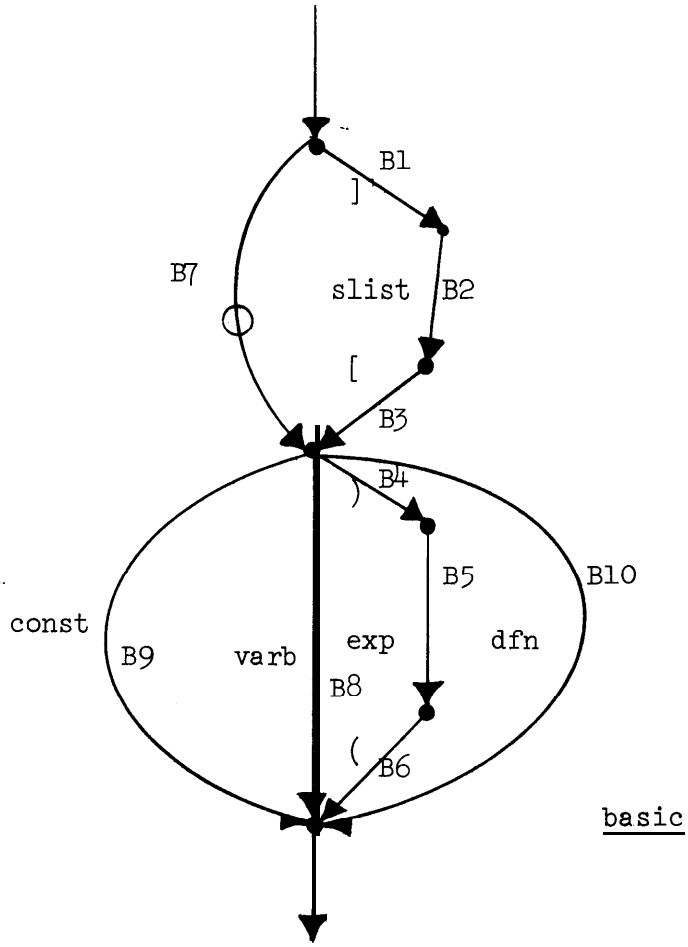
14. Weizenbaum, J., "Symmetric List Processor," Comm. A.C.M., 6, 9 (1963) 524-544.

15. Collins, G.E., REFCO III, A Reference Count List Processing System for the IBM 7094, IBM Research Division, Research Report RC-1436 (May 11, 1965).

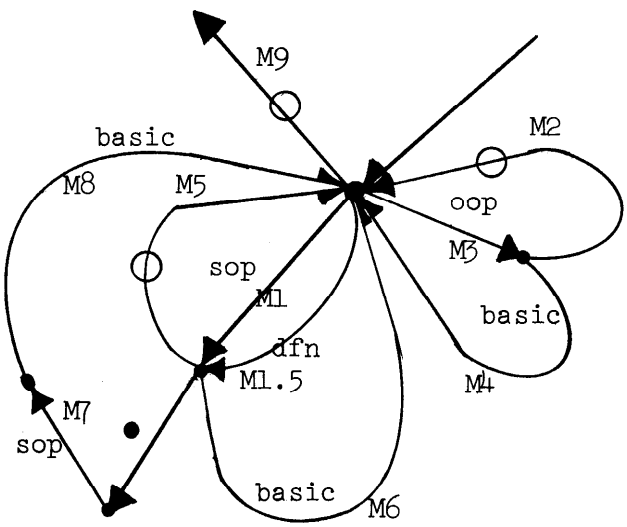
A P P E N D I C E S

APPENDIX A -- Transition Diagrams and their Internal Representation

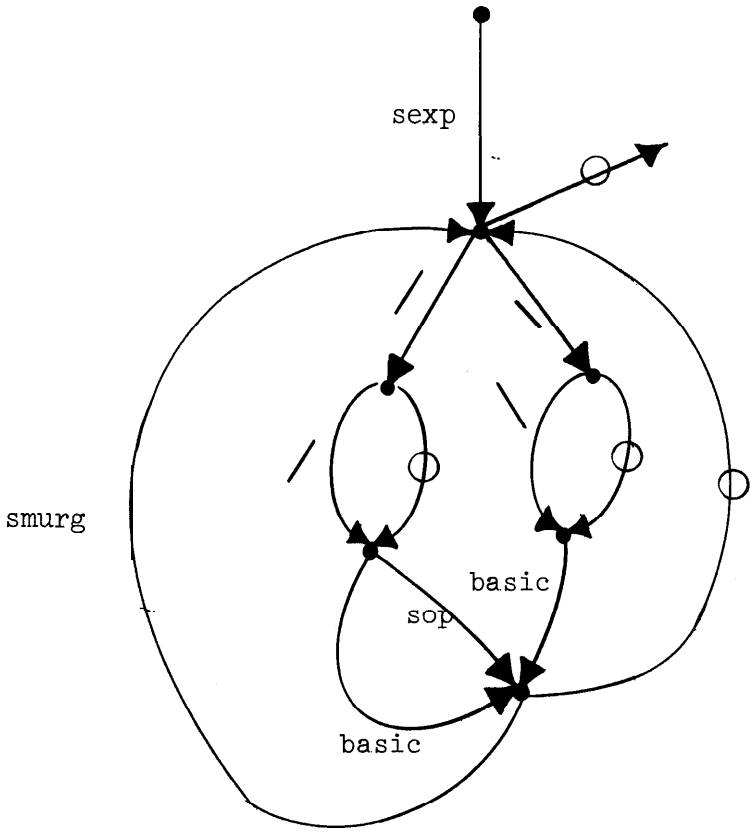


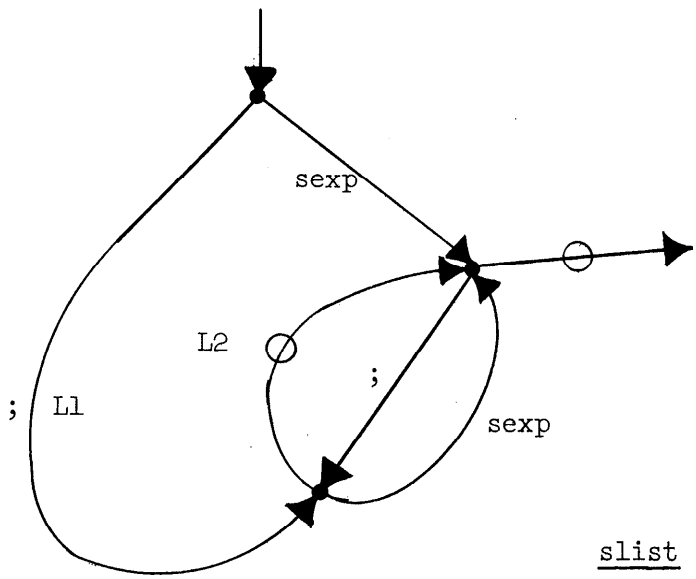

B1
]
B7
slist
B2
[
B3
)
B4
B5
B10
const
B9
varb
exp
dfn
B8
(
B6

basic


M9
basic
M2
M8
M5
oop
M3
sop
M1
basic
dfn
M1.5
M4
M7
sop
basic
M6

smurg


sexp
smurg
basic
sop
basic
exp


sexp
L2
;
;
L1
sexp
slist

The transition diagrams of the preceding pages are stored in the array DIAG. In SYNTAX, DIAG is indexed by D . Each node in a diagram is represented internally by a sequence of triples of words, each of which corresponds to a path from that node. For a given node, if all paths leading from it contain terminal or nonterminal symbols (that is, no null paths from this node), a word containing the flag '1' follows the set of triples for that node.

For each triple, the words have the following contents:

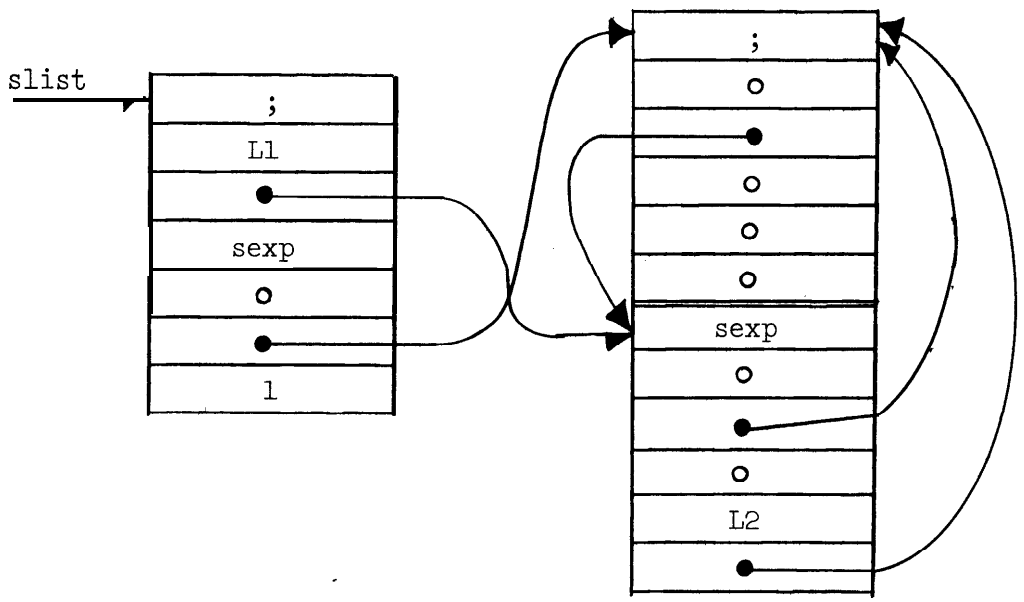
word 1 Class of the element to be scanned for this path, as follows:

0 default path (always satisfied)
1 no more paths from this node
5,6,..., 30 terminal symbol
> 30 index in DIAG of the diagram for a nonterminal symbol to be scanned

word 2 Interpretation rule to be executed if this path is successful

word 3 Link to next node in diagram. If this word is 0, then the diagram has been satisfied.

Example: Schematic of internal representation of the diagram for an slist.



APPENDIX B -- Subprograms in the Interpreter

Program	Function
ADDSYM	Used by TYPEIN to create a new S entry
ARTHTP	Checks for type compatibility and finds common types for operator execution
CODE	Emits one code syllable into high M
DOOOP	oop execution
DOSOP	sop execution
ERROR	Error analysis and recovery
EXC	Execution of individual sops on 1 or 2 scalars
EXPAND	Unpacks 6-bit bytes into 8-bit bytes (written in MAP)
FUZZY	Contains floor, ceiling and approximate comparisons (in MAP)
GCOL	Garbage collector
GETSPA	M storage allocation
GETTEM	Create temp S entry
INCHAR	Character-by-character input routine
INDEX	Subscript execution
MASTER	Main program -- alternates between TYPEIN and SYNTAX
MATRIX	Generalized matrix product execution
OUT	Output routine (used by □ and diagnostics)
PUSH	Stack entry on SV and check for overflow
PUTTEM	Reclaim temp S entry and mark used M space as garbage
SELECT	Compression and expansion execution
REDUCE	Reduction execution
STNEXT	Symbol table search

Program	Function
SYNTAX	See Chapter III
TYPE	Contains fetch and store routines and type conversion (written in MAP)
TYPEIN	See Chapter III
XRHO	Computes $X/\rho A$ for an M entry A

APPENDIX C -- Examples of Programs Run Under the Interpreter

```
                    * FACTORIAL FUNCTION, DEFINED RECURSIVELY
                    *
                    DEFINE Z = FACT N
(   1)              Z = 1
(   2)              GOTO (N EQ 0)/0
(   3)              z = N • FACT   N-I
(   4)              DEFINE
                    •
                    ***  = FACT 3
     6
                    •
                                  T=0.0001
                    • # IS  THE NTH ROOT OF P
                    •
                    DEFINE A = N ROOT P
(   1)              A = P
(   2)              LO.. I = N
(   3)              Z = 1
(   4)              L6.. G O T 0 (I EQ 0)/L2
(   5)              z = A*Z
(   6)              I = I - I
(   7)              GOTO L6
(   8)              L2.. GOT0 IT GE ABS P-Z)/0
(   9)              A = A + (P-Z)DIV N • Z  DIV  A
(  10)              GOTO LO
(  11)              DEFINE'
                    *
                    BOX = 2 ROOT  5
          2.236069
                    BOX = 3 ROOT 27
          3.000000
                    •
                    *PERPETUAL  CALENDAR
                    *
                    DEF INE DAY = CALENDAR D
I   1)              NAMES=(7,4) RHO 'SUN MON TUESWEDSTHURFRI SAT '
(   2)              SUBSCRIPT = 1 + 7 MOD (0,3,3,6,1,4,6,2,5,0,3,5) $(D$(1$)$) H Y P H E N
(   2)                +D$(2$) + (6- 2 • 4 MOD FLOOR D$(3$) DIV 100)   HYPHEN
(   21                + (FLOOR 1.25 *  100  M O D D$(3$))- (D$(1$) LE 2)   HYPHEN
(   2)                AND 0 EQ 4 MOD D$(3$)
(   3)              DAY = NAMES$(SUBSCRIPT ., $)
(   4)              DEFINE
                    *
                    BOX = CALENDAR 12,24,1943
FRI
                    BOX = 'TODAY IS ' , CALENDAR 5,30,1966
TOOAY IS MON
                    *
                    *CALCULATE PASCAL TRIANGLE
                    *
                    DEF INE P A S C A L
(   1)              P = 1
(   2)              BOX =  P
(   3)              P = (0,P) + P,0
(   4)              G O T 0 (N G E P$(2$))/2
(   5)              DEFINE
                    *
                    GENEX - 0
                    •
```

```
                *A FUNCTION OF NO PARAMETERS (THE GENSYM OF LISP 1.5)
                •
                DEFINE X = GENSYM
(   1)          GENEX = GENEX+1
(   2)          X = 'GEN', '0123456789'$(1 + (3 R H O 10) R E P GENEX$)
(   3)          DEFINE
                •
                DEFINE PASCAL1 M
(   1)          N = M
(   2)          PASCAL
(   3)          DEFINE
                •
                *POSITIVE OR NEGATIVE PASCAL TRIANGLE
                *
                DEFINE  FB PASCAL2 N
(   1)          P = 1
(   2)          BOX = P
(   3)          P = (P,0) + FB*0,P
(   4)          G O T 0 (N G E ABS P$(2$))/2
(   5)          DEFINE
                •
                *
                BOX = GENSYM, GENSYM
GEN002GEN001
                A = GENSYM
                BOX = GENSYM
CEN004
                PASCAL1 3
 1
      1     1
      1     2      1
      1     3      3     1
                   (-1) PASCAL2 3
 1
      1    -1
      1    -2      1
      1    -3      3    -1
                *
                FINISH

TOTAL TIME USED        12.472 SECONDS              313 STATEMENTS EXECUTED
        0 ERRORS         0 GARBAGE COLLECTIONS     589 CALLS ON GETSPA
```

```
                    *EXPRESSION PROCESSOR --
                    •  INFIX TO REVERSE POLISH
                    *  REVERSE POLISH TO COMPLETELY PARENTHESIZED
                    DEFINE X = REST Y
(    1)              X =(NOT(RHO Y)ALPHA1)/Y
(    2)              DEFINE
                    • ARITHMETIC EXPRESSION TO POLISH
                    DEFINE S = POLISH I
(    1)              PRI= 0,0,1,1,1,2,2
(    2)              DPS= '$(+-)*/'
(    3)              S= 1 RHO '$'
(    4)              L1.. GOTO(0 EQ RHO I)/L6
(    5)              T = I$(1$)
(    6)              I = REST I
(    7)              GOTO(AND/T NE OPS)/L5
(    8)              TP= (T EQ OPS)/PRI
(    9)              L2.. GOTO((T EQ '(')OR TP GT (S$(1$)EQ OPS)/PRI)/(L3,L4)$(1+T EQ ')'$)
(   10)              S = ROTL S
(   11)              GOTO L2
(   12)              L3.. S = T,S
(   13)              GOTO L1
(   14)              L4.. S = REST S
(   15)              GOTO L1
(   16)              L5.. S = S,T
(   17)              GOTO L1
(   18)              L6.. S = REST(((,'$'EQ S)/IOTA'RHO S)-1)ROTC S
(   19)              DEFINE
                    * POLISH TO FULLY PARENTHESIZED
                    DEFINE 0 =PARENS I
(    1)              S = 0 RHO 'A'
(    2)              01.. T = I$(1$)
(    3)              I = REST I
(    4)              GOTO (AND/T NE '+-*/')/02
(    5)              T = '(' , NEX , T , NEX ,')'
(    6)              02.. S = T,'$',S
(    7)              GOTO(ONE RHO I)/01
(    8)              0 =(SN E'$')/S
(    9)              DEFINE
                    * NECESSARY TO STACK AND UNSTACK STRINGS
                    OEFINE X = NEX
(    1)              R = ((S EQ '$')/IOTA RHO S)$(1$)
(    21              X =((RHO S)ALPHA R-1)/S
(    3)              S = (NOT (RHO S)ALPHA R)/S
(    4)              DEFINE
                    • DRIVER PROGRAM
                    DEFINE PROG I
(    1)              BOX = 'INPUT EXPRESSION...' , I
(    2)              I = POLISH I
(                    BOX = 'REVERSE POLISH...' , I
(    3)4)            BOX = 'FULLY PARENTHESIZED...' , PARENS I
(    5)              DEFINE
                    •
                    * TEST CASES
                    •
                    PROG 'A+B*3/4-2'
INPUT EXPRESSION...A+B*3/4-2
REVERSE POLISH...AB3*4/+2-
FULLY PARENTHESIZED...((A+((B*3)/4))-2)
```

```
                PROG '(A+B)*3/(4-2)'
INPUT EXPRESSION...(A+B)*3/(4-2)
REVERSE POLISH...AB+3*42-/
FULLY PARENTHESIZED...(((A+B)*3)/(4-2))
                PROG '1/(((((A)))))-3*(7))'
INPUT EXPRESSION...1/(((((A)))))-3*(7))
REVERSE POLISH...1A37*-/
FULLY PARENTHESIZED...(1/(A-(3*7)))
                FINISH

TOTAL TIME USED          21.804 SECONDS              639 STATEMENTS EXECUTED
        0 ERRORS           2 GARBAGE COLLECTIONS    1814 CALLS ON GETSPA
```

```
                    • A SIMPLE COMPUTER -- BASED ON NOTES BY IVERSON AND FALKOFF
                    *
                    * REGISTERS       RHO        FUNCTION
                    •      A          16         ACCUMULATOR
                    •      I          16         INSTRUCTION REGISTER
                    •      P          16         PSW
                    •      M        1024,16      MEMORY
                    •
                    • INSTRUCTION FORMAT . . .      XXXXAAAAAAAAAAAA
                    •     WHERE XXXX = OP CODE
                    •           A..A = ADDRESS
                    •
                    •  MNEMONIC CODE FUNCTION
                    •     LD     0010 LOAD
                    •     ST     0001 STORE
                    •     AD     0110 ADD
                    *     su     0101 SUBTRACT
                    •     BU     1001 BRANCH UNCONDITIONAL
                    •     BC     1000 BRANCH CONDITIONAL (IF OVERFLOWJ
                    *     WR     1101 WRITE
                    •     HLT    1111 HALT
                    *
                    DEFINE MACHINE
(    1)             FETCH.. I=M$((BASE(16 OMEGA 12)/P) • ) $)
(    2)             EA = BASE (16 OMEGA 12)/I
(    3)             P$(4+IOTA 12$) = (12 RHO 2) REP 1 • BASE (16 OMEGA 12)/P
(    4)             • TRACE WHAT IS GOING ON INSIDE THE MACHINE
(    4)             BOX = 'P = ',('01'$(1+P$),'      A = ',('01'$(1+A$),'      I = ',('01'$(1+I$)
(    5)             GOTO (LS,AS,BR,IO)$(1+BASE I$(1,2$)$)
(    6)             *LOAD AND STORE
(    6)             LS.. GOTO I$(3$)/LL
(    7)             M$(EA .,$) = A
(    BJ             GOTO FETCH
(    9)             LL.. A = M$( EA .,$)
(   10)             GOTO FETCH
(   11)             *ADD AND SUBTRACT
(   11)             AS.. K1 = BASE A
(   12)             K2 = BASE M$( EA .,$)
(   13)             GOTO I$(4$)/SS
(   14)             K = K1+K2
(   15)             GOTO SA
(   16)             SS.. K = K1-K2
(   17)             SA.. A = TWOS REP K
I  18)              P$(1$) = ((17 RHO 2) REP KJ $(1$)
(   19)             GOTO FETCH
(   20)             *BRANCH
(   20)             BR.. GOTO (NOT P$(1 + B A S E I$(3,4$)$)   J/FETCH
(   21)             P$(4 + IOTA 12$) = (16 OMEGA 12)/I
(   22)             GOTO FETCH
(   23)             10.. GOTO I$(3$)/0
I  24)              BOX = '---OUTPUT---    0 , '01'$(1 + M$( EA .,$)$)
(   25)             GOTO FETCH
(   26)             DEFINE
                    *
                    *HANDY CONSTANTS FOR M SETUP
                    B = 12 RHO 2
                    TWOS = 16 RHO 2
                    X0 = 12 RHO 0
                    X100 = B REP 100
```

```
X101 = B REP 101
x102  = B REP 102
X103 = B REP 103
X104 = B REP 104
LD = 0,0,1,0
ST = 0'0'0'1
AD = 0,1,1,0
su = 0,1,0,1
WR = 1,1,0,1
HLT = 1'1'1'1
BU = 1'0'0'1
BC = 1'0'0'0
M = (1024,16) RHO 0
A = 16 RHO 0
I = A
P = ROTL 16 ALPHA 5
• 'LOAD' PROGRAM INTO M
• THIS PROGRAM HULTIPLIES M$(101.,$) BY M$(100.,$) AND PRINTS THE RESULT
M$( 1  • ' $) = LD,X103
M$( 2  ., $) = ST,X102
M$( 3  ., $) = LD,X100
M$( 4  ., $) = SU,X104
M$( 5 ., $) = BC,B REP 1 1
M$( 6 ., $) =  ST,X100
M$( 7  ., $) = LD,X102
M$( B ., $) = AD,X101
M$( 9 ., $) = ST,X102
M$(10 .,$) = BU,B REP 3
M$(11  ., S J = WR,X102
M$(12 ., SJ = HLT'XO
M$(104.,$) = 16 OMEGA 1
•
• MULTIPLY 1000 TIMES 4
•
M$(100 .,$) =  x0,0,1,0,0
M$(101 ., $) = 116  RHO 2) REP 1000
•
MACHINE
P = 1111000000000010  A =  0000000000000000     I = 0010000001100111
P = 1111000000000011  A =  0000000000000000     I = 0001000001100110
P = 1111000000000100  A = 0000000000000000      I = 0010000001100100
P = 1111000000000101  A =  0000000000000100     I = 0101000001101000
P = 0111000000000110  A =  0000000000000011     I = 1000000000001011
P = 0111000000000111  A=   0000000000000011     I = 0001000001100100
P = 0111000000001000  A =  0000000000000011     I = 0010000001100110
P = 0111000000001001  A =  0000000000000000     I = 0110000001100101
P = 0111000000001010  A =  0000001111101000     I = 0001000001100110
P = 0111000000001011  A=   0000001111101000     I = 1001000000000011
P = 0111000000000100  A =  0000001111101000     I = 0010000001100100~
P = 0111000000000101  A =  0000000000000011     I = 0101000001101000
P = 0111000000000110  A =  0000000000000010     I = 1000000000001011
P = 0111000000000111  A =  0000000000000010     I = 0001000001100100
P = 0111000000001000  A =  0000000000000010     I = 0010000001100110
P = 0111000000001001  A =  0000001111101000     I = 0110000001100101
P = 0111000000001010  A =  0000011111010000     I = 0001000001100110
P = 0111000000001011  A =  0000011111010000     I = 1001000000000011
P = 0111000000000100  A =  0000011111010000     I = 0010000001100100
P = 0111000000000101  A =  0000000000000010     I = 0101000001101000
P = 0111000000000110  A =  0000000000000001     I = 1000000000001011
P= 0111000000000111     A = 000000000000000~    I = 0001000001100100
```

```
P = 0111000000001000      A = 0000000000000001      I = 0010000001100110
P = 0111000000001001      A = 0000011111010000      I = 0110000001100101
P = 0111000000001010      A = 0000101110111000      I = 0001000001100110
P = 0111000000001011      A = 0000101110111000      I = 1001000000000011
P= 0111000000000100       A= 0000101110111000          0010000001100100
P= 0111000000000101       A= 0000000000000001      I = 0101000001101000
P= 0111000000000110       A= 0000000000000000      I = 1000000000001011
P= 0111000000000111       A= 0000000000000000          0001000001100100
P= 0111000000001000       A= 0000000000000000      I = 0010000001100110
P= 0111000000001001       A= 0000101110111000      I = 0110000001100101
P = 0111000000001010      A= 0000111110100000      I = 0001000001100110
P = 0111000000001011      A= 0000111110100000      I = 1001000000000011
P = 0111000000000100      A= 0000111110100000          0010000001100100
P = 0111000000000101      A = 0000000000000000      I = 0101000001101000
P = 1111000000000110      A = 1111111111111111      I = 1000000000001011
P = 1111000000001100      A= 1111111111111111       I = 1101000001100110
---OUTPUT---    0000111110100000
P = 1111000000001101      A = 1111111111111111      I = 1111000000000000
                FINISH

TOTAL TIME USED          34.622 SECONDS              378 STATEMENTS EXECUTED
        0 ERRORS           4 GARBAGE COLLECTIONS     1730 CALLS ON GETSPA
```