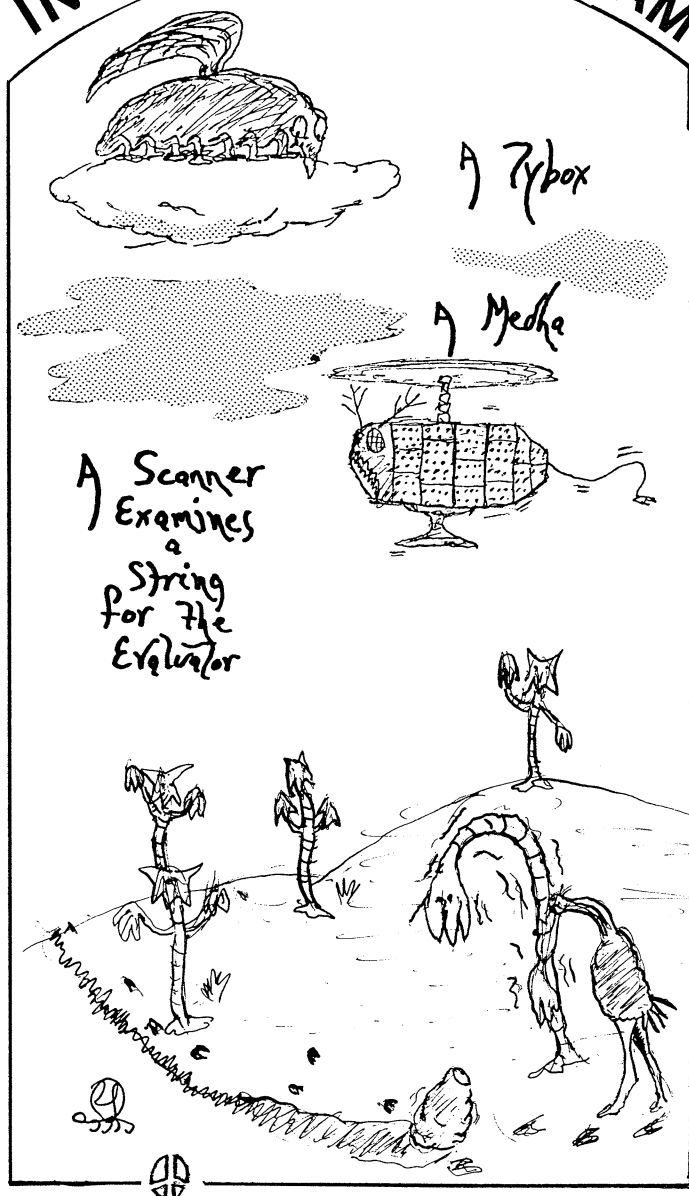


SAM76

the first LANGUAGE manual

IN THE LAND OF SAM



Second Edition

The SAM76 Language

The SAM76 language was designed by people for people - not by programmers for programmers. It follows a well defined syntax which is easy to learn and to read. The notation avoids the use of pseudo "English" words which are a frequent source of confusion and ambiguity in many of the other computer languages.

The SAM76 language can be used in as large a variety of tasks as one is able to imagine - this on personal computers without requiring computer specialists or programmers to intercede.

There are more than 150 functions - or instructions - available making the SAM76 language the most powerful available today, and it fits in approximately eight thousand bytes of memory; this can be ram or rom as the user desires.

The SAM76 language can be viewed as a real language which follows the user's stream of consciousness in much the same manner as spoken language. This permits the language in its written form as used by the computer and the user to serve as documentation.

The SAM76 language provides the user with the capability of requiring the computer to perform complex operations in many areas; a few of these are: Control, Text manipulation and editing, Simulation, Arithmetic with any desired precision.

The SAM76 language is interactive and reactive. As one task is accomplished the user continues and in effect the SAM76 language processor carries on a conversation, reacting to expressed desires.

The SAM76 language provides a uniquely flexible means to control facilities or to derive data from sources other than the user's keyboard.

The SAM76 language is a "string processor". This means that the units of information are not confined to any fixed length, but may be made up of any number of characters, or even no characters, as determined by the user. Entire strings may be manipulated by single commands.

The SAM76 language is interpretive. This means that when a string is evaluated and an expression found to contain an instruction or command, then the specified action is immediately performed and the resulting value, if any, replaces that expression in the string.

The SAM76 language facilitates the use of pre-defined procedures. This means that the user's procedures or scripts may be stored for potential use and later called by name and immediately acted upon, with variables supplied to specified arguments as part of the process.

The SAM76 language makes no distinction, except in the user's own use of information, between data and procedures. Procedures tell the processor what to do; data is the information acted upon by the procedures. Procedures may be modified when other procedures treat them as data.

The SAM76 language is most powerful in providing man-machine interaction permitting the user to modify his work and to intervene when desired. The language provides facilities to define and save scripts for subsequent use; this in effect can behave or operate as if they themselves were inherent functions of the language.

designed for you and your personal computer

101	-	os,s1	Output String
102	-	is,dev1	Input String
103	-	dt,t,s,d1,d21	Define Text (superseding)
104	-	et,t1,t2,...,t1	Define Text (save current)
105	-	et,t1,t2,...,t1	Erase Text
106	-	et,t1,t2,...,t1	Erase all occurrences of Text
107	-	lt,s0,d1,d2,...,d1	List Texts
108	-	lt,t,s1,s2,...,s1	Fetch Text
109	-	pt,t,s1,s2,...,s1	Partition Text all matches
110	-	pt,t,s1,s2,...,s1	Partition Text next match
111	-	pc,t,s1,s2,...,s1	Partition Character
112	-	mt,t,s1,s2,...,s1	Multi-part Text all matches
113	-	mt,t,s1,s2,...,s1	Multi-part Text next match
114	-	mc,d	Multi-partition Character
115	-	ni,yt,vf1	Neutral Implied
116	-	ex,f	Exit
117	-	ca,s	Change Activator (current)
118	-	ca,s	Change Activator (initial)
119	-	ht,t1	Hide Text
120	-	id,d1	Hide all Texts
121	-	id,s1,s2,...,s1	Input Character
122	-	vt,t1,t2,...,t1	Input 'D' characters
123	-	xr,x1	Input to Match
124	-	xw,x1	View Texts
125	-	xrp,x1	examine Register
126	-	xwp,x1,x21	examine Register Write in reg.
127	-	xj,x1	examine Register Pair
128	-	tma	experimental Write reg. Pair
129	-	tm,d1	experimental Jump
130	-	yt,t,s,vt,vf1	Trace Mode All activated
131	-	ad,n1,n2,n3,...,n1	Trace Mode All deactivated
132	-	su,n1,n2,...,n1	Trace Mode activated
133	-	ct,t1,t2,t3,...,t1	Trace Mode deactivated
134	-	cnb,d1	ys There
135	-	gnb1	Text Branch
136	-	ig,d1,d2,vt,vf1	Add
137	-	fg,t,vz1	Subtract
138	-	fde,t,d,vz1	Multiply
139	-	fde,t,d,vz1	Divide
140	-	fdm,t,d,s,vz1	Combine Texts (superseding)
141	-	fe,t,vz1	Combine Texts (save current)
142	-	ff,t,d,vz1	Change Number Base (active)
143	-	fi,t,s,vz1	Change Number Base (initial)
144	-	fr,t,s,vz1	Query Number Base
145	-	fp,t,x1,...,x1	If Greater
146	-	md,t,d1	Fetch Character
147	-	ord,t1	Fetch "D" Characters
148	-	cld,t1	Fetch "D" Elements
149	-	hp,t,d1	Fetch "D" Matches
150	-	hm,t,s1	Fetch Element
151	-	ep,t,p1,p2,...,p1	Fetch Field
152	-	it	Fetch Left match
153	-	idt,d1	Fetch Right match
154	-	ot,t1,t2,...,t1	Fetch Partition
155	-	ot,leader,gap,t1,t2,...,t1	Move Divider to pos. "d"
156	-	x11,s01	Move Divider "d" increments
157	-	xal,label,offset1	Characters Right of Divider
158	-	sfd,fun dev1	Characters Left of Divider
159	-	sar1	How many Partitions
160	-	ab,s1,s2,vt,vf1	Erase Partitions
161	-	ai,s0,s1,s2,...,s1	Erase Text
162	-	as,s0,s1,s2,...,s1	Input Text
163	-	ps,d,s1,s21	Input "D" Texts
164	-	rs,s1	Output Texts
165	-	ds,d,s1	Xamine Label List
166	-	rr,s11	Xamine Address of Label
167	-	gp,t1	Specify Function Device
168	-	tr,t,s1	Auto Return on line feed
169	-	ut,cc1	no Auto Return on line feed
170	-	xc,x1,x2,...,x1	Alphabetic Branch
171	-	cx,s0,s1	Alphabetic Insertion
172	-	dx,d,x1	Alphabetic Sort
173	-	pl,sub,s11,...,s1	Pad String
174	-	w1,xn1,yn11	Reverse String
175	-	wx	Duplicate String
176	-	wy	Return to Restart
177	-	wz	Restart Initialized
178	-	w1	Query Partition
179	-	ws,xn1,yn1,...,xn,yn1	Trim
180	-	wc,s1,s1	User Trap active
181	-	zd,r,v-,v0,v+	User Trap inactive
182	-	zi,r,v-,v0,v+	X to Character
183	-	zq,r1	Character to "X"
184	-	zs,r,n1	"X" to Decimal
185	-	or,x1,x21	Plot
186	-	and,x1,x21	Write Initialize
187	-	not,x1	Write "X" displacement
188	-	rot,d,x1	Write "Y" displacement
189	-	sh,d,x1	Width Right
190	-	cil,d1	Width Left
191	-	qil1	Write Straight Lines
192	-	cin,t1,d1,...,t,d1	Write Characters
193	-		"Z" reg. Decrement and branch
			"Z" reg. Increment and branch
			"Z" reg. Query
			"Z" reg. Set
			Or the bits
			And the bits
			Not (complement) the bits
			Rotate the bits
			Shift the bits
			Change Line length (active)
			Change Line length (initial)
			Query Line Length
			Change Id Number

The SAM76 Language Handbook

Ancelme Roichel

and many others



DISKLAIMER and Copyright

DISKLAIMER: HAVING RECIEVED THE MANUSCRIPT FOR THIS BOOK IN THE FORM OF A REEL OF MAGNETIC TAPE WHICH THE AUTHORS ASSURED US WAS OF THE MOST COMPATABLE KIND, WE CANNOT ASSUME ANY RESPONSABILITY WHATSOEVER FOR YNY ERRORS THE READER MAY ENCOUNTER AS A RESULT OF THE WORK (INCLUDING ALL EXAMPLES, BUT EXCLUDING THIS DISKLAIMER) HAVING BEEN PROCESSED BY A COMPUTER.

THE PUBLISHERS

First Edition - March 1978 - 500 copies as follows:

- 10 - in-folio - numbered I to X each accompanied
by one of Joseph Tulloch's original drawings.
- 190 - in-folio edition for general sale
- 300 - "perfect" bound for general sale

Second Edition - January 1979 - 1000 copies in folio; this edition corrects a number of small errors in the first printing and includes additional material which was made available through updates and publications in Dr. Dobbs.

Copyright (c) 1978, 1979 by Ancelme Roichel

Portions of this work was derived with permission from earlier writings by members of the R.E.S.I.S.T.O.R.S.

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without written permission of the publisher.

Dedication

A SAM76 LANGUAGE HANDBOOK

and

Other goodies

dedicated to

Barry Klein, David Theriault, Nat Kuhn, John Levine

and

their renowned artist

Joseph Tulloch

as well as to

Karl Nicholas

who put the finishing touches to this work

and finally brought it to press

Acknowledgements

Introduction of the SAM76 language during the course of 1976 and its availability on micro Computer systems prompted the production of this book; portions of it were derived from material written some years ago by several members of a group of youngsters calling themselves the R.E.S.I.S.T.O.R.S.

Thanks are extended to the various people who contributed to the development coding and check out of SAM76. Among these are Neil Colvin, Marty Nichols, Carl Galletti, Tom Kirk, Roger Amidon, Doug Moser, Allen Katz, many members of the Amateur Computer Group of New Jersey, and last but not least New Jersey's rapidly growing network of Computer Stores. We also thank Jim Warren of Dr. Dobb's Journal of Computer Calisthenics and Orthodontia as well as Carl Helmers of Byte magazine who provided the final encouragement.

Particular thanks are extended to Barry Lubowsky and Rider College for making available computer time on their PDP-10 to produce the SAM76 language processor and to edit this book.

Lastly it was felt appropriate to reproduce below the "acknowledgments" in the R.E.S.I.S.T.O.R.S. own work which never reached the publication stage.

Ancelme Roichel - Winter 1978

The Authors, although having written a book, do not deserve sole recognition for this work. Many people were involved both helping and hindering the production of it, but all with good intentions.

R.E.S.I.S.T.O.R.S.
Acknowledgements

We would particularly like to thank:

Claude Kagan for finding us contacts, flogging us on, and having so many ideas that we dared not implement them all.

A very special man (whose name we dare not mention) who helped us understand the subtleties of another very interesting computer language and brought us ice cream that melted during a discussion.

Chuck Ehrlich who picked up the pieces after we got fed up with our book and found more fun things to do like college.

Giff Marzoni for sparing us the arduous task of typing late at night.

Ginny Berthold who worked after hours to typeset the manuscript.

Other people to whom we are greatly appreciative:

Alan Taylor, Mrs. Taylor who bought extra blankets so we could stay over at her house during a hectic editing session, Tony Weber who helped us with equipment operation - maintenance - and use, Jim Gorman for the use of his microbus and the contribution of limericks, Perry Hess for his help in better understanding, Jerome N.B. King, Andy Huson, Reid Anderson, John Kirkley, Dan Scott, Herb Grosch who was photographed speechless when confronted with Nat, Larry Laitinen who kept busy keeping the model 35 Teletype running.

Also Dave Hanson who scrolled many lines of diagrams for us, Jack Kinn of IEEE who wrote us to try the A.C.M. for information about University Computer Science possibilities, Bob Leach, Hank Marrows who did P.R. for us as well as for his company, Ed Pellman who helped us get started with the loan of a computer, Fred Hatfield, Margaret Fox who made a cameo appearance in a movie we started to make and never finished, Larry Schear, Jeff White who while collecting spelling mistakes for a study of his - corrected ours.

And then to the people at Alcom, Calcomp, Raytheon, various Bell System companies, Tektronix, Digital Equipment Company, Computerworld, Brandon Publishing Co. as well as to too many others whose names we forgot, the RESISTORS Past and Present and of course the little PDP-8.

And special thanks to the people who had trust enough in us to buy a copy of our book before it was ever finished:

Murray Freeman, Richard Gardner, John Hamilton, Elizabeth Becker, John Nagle, Melvin Fisher, Jim Orcutt, Mark Goldstein, Eric Clamons, Marge Hill, Neil Smith, Roger Van Ghent, Herbert Kukash, Douglas Eastwood, Walter Daughterity, Ellen Wax, Lester Corrsin, Joel Rose, E. B. Smith, Martin Cornelius Jr., Mark Bayern, Doug Grant, Joe Hilsenrath, Ken Weiner, Greg Williams, Jerry Ogdin, Henri Socha, Tom Ziehe, Gary Coleman, Alan Lemmon, David Beaucage, Thomas Giventer and Michael Lowery .

Thank you one and all,

Barry, Dave, Nat, John and Joe

Summer 1971

FOREWORD

by Claude A. R - Kagan, RESISTORS' Adviser

This piece of interesting reading was put at the back of the book so that it would not distract you from what you bought the book for. Now, without further delay, you may begin to learn about the SAM76 language.



LESSON ZERO

To Those Inexperienced in the Use of Computers

The computer language you are about to learn will enable you to communicate with and use a computer, but before you leap "into the SAM76 language" a small introduction to computers and associated equipment is in order. The piece of equipment that the user of the SAM76 language uses the most is the terminal. This is a device which differs from an electric typewriter only in that when a key is hit on the terminal, an electrical code signal is sent to the computer. The computer can also send a code signal back to the terminal, causing the terminal to display (on paper or a TV screen) a character, or symbol. In this way, a person and a computer can "talk" to one another.

The SAM76 language combines and extends the features of earlier languages, in particular:

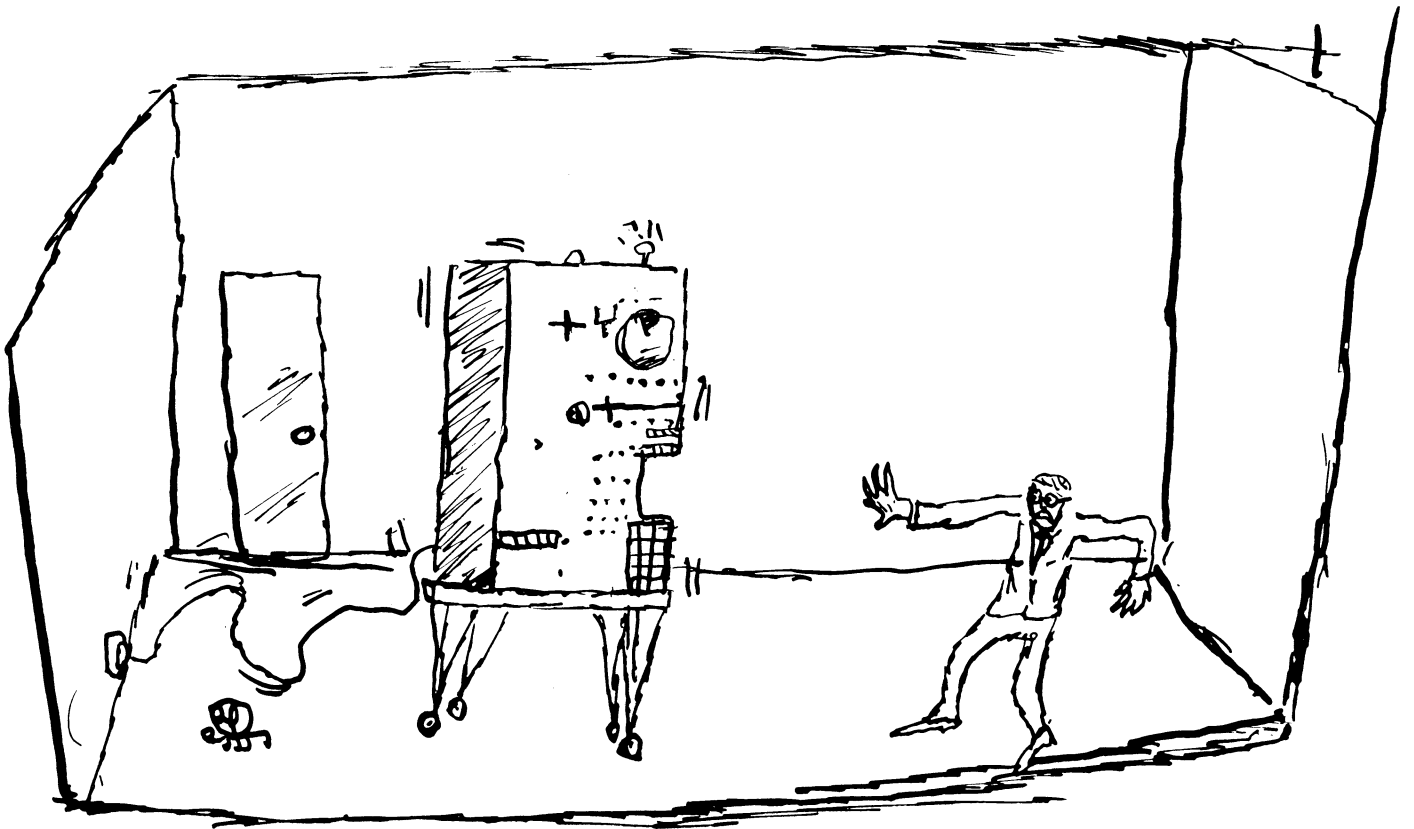
GPM "General Purpose Macro Generator" by STRACHEY
and
M6 by Doug. Mc Ilroy

and gives you, the user, a method of performing arithmetic operations in the manner taught you in school.

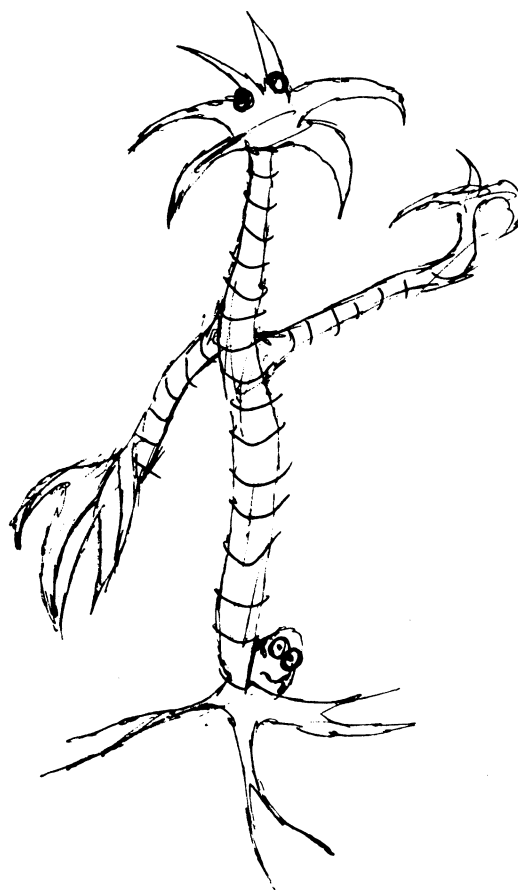
The SAM76 language works with text. Text is anything that can be typed on the terminal keyboard. A term that you will meet in dealing with text is the word "string". Since a computer reads the codes one by one, text really is a string or sequence of characters, or symbols.

Thus, in the following book, text strings are what we usually consider.

With this very sketchy background you can feel secure in reading the following volume on the SAM76 language.



[illegible]



LESSON ONE - Syntax

The SAM76 language deals with sequences of characters. You might ask: "What is a character? - Is a space a character? - Am I a character?"; well, by character we mean any single item of information which can be represented in the computer by some combination of bits, and which generally can be sent from the user terminal to the computer by depressing a single key (or combination of keys - as when you hold the shift key down to get upper case "characters"; this includes, in this manner, all of the symbols that get printed, the non printing codes which cause the terminal to do things like going to the beginning of the next line (which we shall call "new line", or "line feed"), or ring the bell, feed paper out, shift ribbon color and so on...

```
|@B4I4QRU/18L#|
```

It sees importance in certain combinations of characters.

```
{ }-----
{ } %AD,4,5/={9}
{ }
-----
```

This illustrates a complete SAM76 language expression. This expression adds four and five and gets nine, with the nine being output by the computer.

Before we forget, we should tell you about some of the symbols used in the illustrations:

```
| ..... | Subject of discussion between vertical lines
{ ..... } Output - sometimes said to be in "red".

{ }-----
{ } Suggests output on paper
{ }
-----
```

The SAM76 language expressions must contain specific patterns. The above expression is an example of the simplest type of a SAM76 language expression which is called a primitive.

This primitive or primitive expression is divided into the following patterns:

|%| or |&|

Either of these two characters, percent or ampersand, signify to the computer that an expression is starting.

|AD,4,5|

This part of the expression tells the computer what to do. AD is the code that tells the computer to add the following two strings of numbers which are separated by commas. There are two commas in this section. As can be seen, the second comma separates the two numbers to be added. The first comma separates the code AD from the numbers to be added. This two letter code tells the computer that the primitive should perform an addition. The three sections divided by commas are called arguments. Commas, when used to separate arguments, are called delimiters. This primitive which uses the two-letter code (mnemonic) AD has three arguments. The first, containing AD, specifies that the primitive should add the second and third arguments, containing four and five respectively.

|/|

A slant sign follows the third argument. A slant sign tells the computer that the primitive expression that started with the last percent or ampersand ends here. This slant sign matches the percent or ampersand before AD.

|%AD,4,5/|

There is one more character that follows the slant sign.

|=|

This character is an equal sign. This equal sign tells the computer that all the material has been typed in and that the computer should now react. The equal sign is called the Activating Character.

All languages of any complexity must have some symbols which give them structural order and unity. In the English language punctuation does this. In the SAM76 language syntax does this. The symbols used above for this purpose are the percent |%| or ampersand |&|, comma |,|, and slant sign |/. These four symbols are therefore called warning characters in the SAM76 language *.

The SAM76 language is by definition capable of handling text; therefore it must contain some text handling primitives.

Text is any character or string of characters that can be typed on the keyboard of a terminal.

```

{}-----
{} %OS,HELLO/={HELLO}
{}
{}-----

```

This is an example of a text handling expression. OS is the mnemonic (code) for Output String. The OS primitive handles text. It outputs the second argument; the text string that follows the comma. As before,

OS
Output String

|%| or |&|

signifies the start of the expression.

|OS|

is the two-letter mnemonic for Output String and is the first argument of the expression. The string

|HELLO|

is the second argument of the expression. It follows the comma (delimiter), and is the text string that will be output.

The next character after the second argument (HELLO) is

|/|

the slant sign. This informs the computer that the idea (primitive, expression) which started with percent or ampersand is complete. The equal sign

|=|

tells the computer that the previously typed character, in this case the slant sign, is the last character of the expression. It also tells the computer that everything is done and that the machine should evaluate the characters that have been typed in. When the computer evaluates the characters it sees that it is supposed to type out

{HELLO}

and therefore HELLO is typed out.

```

{}-----
{} %OS,HELLO/={HELLO}
{}
{}-----

```

This is what the expression would look like if the user of the computer had typed in the expression in black and the computer had typed out the answer in red.

Stumble onward; you'll soon walk.

Note *	The authors are using the warning characters shown above in this book, for the "S" syntax form of the SAM76 language. In addition the "S" syntax makes use of the exclamation mark, parentheses, angle brackets, and the commercial @ "at". The "M" syntax uses the sharp # sign, the : colon, and the ; semicolon.
--------	---



LESSON TWO - Scanning

The order of evaluation is not important when there is only one expression to be evaluated, but in the SAM76 language it is very desirable to evaluate several expressions in a group when the activating character is typed.

This is an example of a multiexpression:

```

{}-----
{}  %AD, 5, 2/%AD, 3, 3/={76}
{}
{}-----

```

When the activating character is typed, the computer starts to evaluate.

|%AD, 5, 2/%AD, 3, 3/|

The example consists of two distinct primitive expressions which must be evaluated. The problem is, which expression to evaluate first? By establishing an order of evaluation it is possible to solve this problem.

The order of evaluation in SAM76 language is expression by expression, from left to right. The mechanism that decides the order of evaluation of expressions is called the scanner.

The scanner reads character by character, from left to right, looking for the first complete expression that can be evaluated. This process is called scanning.

When this example

|%AD, 5, 2/%AD, 3, 3/|

is scanned, the scanner starts to read the characters, looking for the warning characters, especially a slant sign, since this character usually means that a primitive expression has just been completed. Here the first character it sees is the percent sign. The scanner notes this character because it marks the beginning of an expression. The scanner continues reading across until it finds the slant sign which matches the percent sign that initiated the expression. When the slant sign is found, the scanning process halts.

```
|%AD, 5, 2/|
```

This is the primitive expression that has just been scanned. It is now evaluated. The result, called the value, is found to be the digit 7. An interesting event now occurs. Since the value (result) of the expression is 7, 7 and %AD, 5, 2/ are equivalent after evaluation; therefore the expression %AD, 5, 2/ can now be replaced by 7. After this is done, the string that the scanner is looking at is

```
|7%AD, 3, 3/|
```

The scanner continues past the seven and comes to the expression

```
|%AD, 3, 3/|
```

It goes through the same process as before. The value of the expression is found to be 6, so 6 now replaces %AD, 3, 3/.

The value of the multiexpression is 7 and 6 adjacent to each other.

Here is what would be on the teleprinter sheet (user types in black, computer answers in red)

```
{ }-----
{ }  %AD, 5, 2/%AD, 3, 3/={76}
{ }-----
```

After a primitive expression is evaluated and replaced by its value, the scanning process restarts itself and begins looking for the next leftmost expression. Using the ability of the language to replace expressions with their values and then to rescan, very powerful combinations of commands can be written using a technique called nesting.

This is an example of a nested expression:

```
{ }-----
{ }  %AD, %AD, 5, 2/, 4/={11}
{ }-----
```

Notice the way in which one expression is imbedded with another.

When the scanning process begins, the scanner immediately notices the first percent sign but it continues reading across beyond the comma. Suddenly the scanner sees another percent sign. The string being scanned is now marked where the second percent sign is located:

```
|%AD, %AD, 5, 2/, 4/|
```

After marking the second percent sign the scanner continues reading across until it finds the first slant sign. This slant sign matches the second percent sign.


```
|%AD, 5, 2/|
```

This is the primitive expression that has been marked out. It is now evaluated and replaced by its value, 7. The digit 7 now replaces %AD, 5, 2/ in the larger expression, so that what the scanner now sees is:

```
|%AD, 7, 4/|
```

This expression is rescanned, evaluated, and replaced by its value of 11, which is printed out.

The primitive IS, which stands for Input String, is another text-handling primitive. It brings text characters from the keyboard into the computer.

IS
Input String

Here is an example of a complete Input String primitive expression:

```
|%IS/|
```

In the above example, percent sign again indicates that an expression is starting. Following this character is the only argument of an Input String expression, the two-letter mnemonic symbol IS. The expression is completed by the matching slant sign. The equal sign (the activating character) tells that a complete expression has been typed in.

Input String is used to input (read in) a text string of any length from the keyboard. The end of the Input String is indicated by typing the activating character, an equal sign. The string now becomes the value of the Input String. %IS/ is replaced by its value, which is this text string.

An Input String primitive is usually nested inside another expression.

Here is an example of an Input String nested within another expression:

```
|%AD, 5, %IS//|
```

On scanning, the first (inner and leftmost) complete primitive expression is found to be %IS/. This expression is executed and the computer is now waiting for a string terminated by the activating character to be typed at the keyboard. This string can be of any length (any number of characters), including zero length (no characters) and made up of any character or string of characters that can be struck on the keyboard. A string of no characters is called a null string.

In this case, 24 is the string that will be typed.

Since 24 will be typed in, it shall become the value of the Input String and will replace the Input String primitive in the nested expression.

The scanner now sees this expression:

```
|%AD, 5, 24/|
```

It is now scanned and evaluated. The value of the expression is 29, which is now printed out.

This is what the total expression would look like printed out on a teleprinter:

```
{ }-----
{ }  %AD, 5, %IS//=24={29}
{ }-----
```

It is possible, using just these techniques, to nest expressions to an unlimited depth, with the only limit being the size of the computer's memory.

For example, the expression

```
{ }-----
{ }  %AD, 5, %AD, 6, %IS///=5={16}
{ }-----
```

contains three levels of nesting, one primitive expression nested inside another expression with this expression nested within yet another expression.

This expression would be evaluated in a manner very similar to the process just used on the example %AD, 5, %IS//. The scanner reads across, noticing and marking the percent signs, until it finds a slant sign, which temporarily stops it.

```
|%AD, 5, %AD, 6, %IS///|
```

The first slant sign found completes the Input String primitive. The computer is now waiting for the string to be typed in. The digit 5 is typed in. This becomes the value of the Input String, and thus replaces the Input String expression. The scanner will then see this expression:

```
|%AD, 5, %AD, 6, 5/|
```

The scanner again reads until it sees the next slant sign. Upon finding this slant sign it stops.

```
|%AD,6,5/|
```

This primitive expression is now evaluated, and its value, 11, replaces it. This is the expression now being scanned:

```
|%AD,5,11/|
```

The scanning process continues until the finding of another slant sign completes the expression. When this is found, the expression %AD,5,11/ is evaluated and replaced by its value, 16. This value is now scanned, but since there are no more expressions left 16 is printed out.

Here is another example of nesting:

```
|%OS,%IS//|
```

When this expression is scanned, the first complete primitive expression is the Input String primitive %IS/. At this point a string must be typed in. The string HELLO is typed, and the activating character is typed to signal the end of the Input String primitive. %IS/ is now replaced by HELLO, its value.

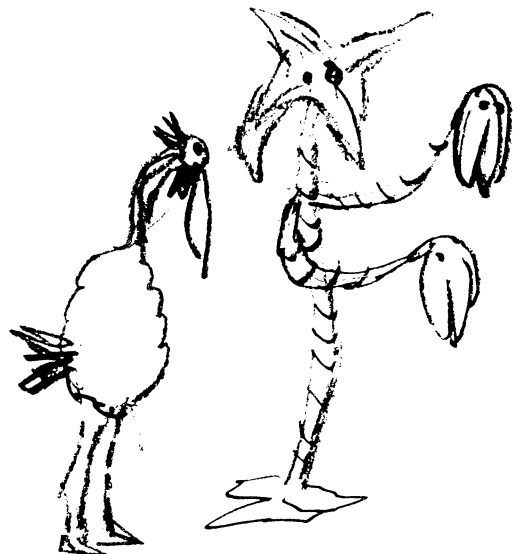
Here is what the scanner now sees:

```
|%OS,HELLO/|
```

This is scanned and the slant sign is found. The complete expression is executed, causing the string HELLO to be printed out.

```
{ }-----
{ } %OS,%IS//=HELLO={HELLO}
{ }
{ }-----
```

The way in which an Input String primitive, which is capable of being replaced by a string of theoretically infinite length, is nested within an Output String primitive, which is capable of printing out any length string, is handled is of utmost importance to the processor. In the following lesson this importance is demonstrated.





LESSON THREE - The Restart Expression

In the preceding discussions only the primitives AD, OS, and IS were mentioned. Of these three primitives only OS has the ability to output characters.

It was just stated that only the primitive OS has the ability to output characters, but there is an inconsistency here. In the earlier lessons, when an add primitive was executed the answer was typed out. If the AD primitive does not have the ability to output characters, how was this done?

The answer is in this nested expression:

```
|%OS,%IS//|
```

In Lesson Two, an example of the above expression was given. This is another example of the nested expression:

```
{ } -----
{ }  %OS,%IS//=%AD,5,2/={7}
{ } -----
```

In this expression evaluation takes place in the following manner. The scanner reads across, noticing the first and then the second percent `|%` sign, until it finds the first slant `|/|` sign, at which point it stops. It starts to evaluate the `%IS/` that it has discovered while scanning. The string `%AD,5,2/` is typed in and replaces the `%IS/`, so what the scanner now sees is:

```
|%OS,%AD,5,2//|
```

At first glance it appears that `%AD,5,2/` should be typed out, but this wouldn't be, following the rules of evaluation which were stated in the preceding lesson.

```
|%OS,%AD,5,2//|
```

When this expression is scanned, both percent `|%` signs are taken note of. Therefore, when the scanner sees the first slant `|/|` sign it stops and evaluates the primitive expression just completed. This is not the OS primitive, but `%AD,5,2/`. This expression is evaluated and its value, 7, replaces it.

|%OS,7/|

This is what the scanner now sees. The last slant |/| sign is scanned and the complete expression %OS,7 is executed, with 7 being printed out.

The final value of an Output String primitive is nothing, or the null string. The scanner is through.

Using the nested function %OS,%IS//, it is possible to get the value of an AD expression printed out. At the beginning of this lesson the question was raised of how the result of an AD expression was typed out. The answer is very simple. Already present in the computer is the nested expression %OS,%IS//.

When characters are typed, as when the expression %AD,5,2/ was typed in, what is really happening is that the %IS/ primitive in the nested expression %OS,%IS// (called the Restart Expression) is replaced by the typed-in string, so that the typed-in string is always nested within an Output String expression.

|%AD,5,2/|

When this expression is typed in, it replaces %IS/ in the Restart Expression %OS,%IS//.

This is what the scanner would actually see if %AD,5,2/ were typed in:

|%OS,%AD,5,2//|

When this is scanned, %AD,5,2/ is recognized, evaluated, and replaced by its value, 7.

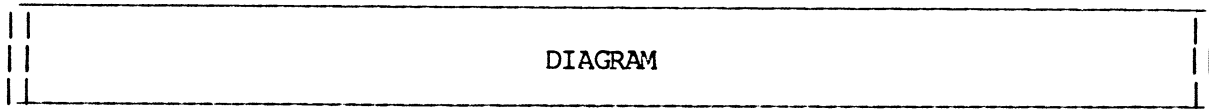
This is what the scanner now sees:

|%OS,7/|

It then executes this expression and 7 is printed out. The OS primitive expression is replaced by its value, the null string.

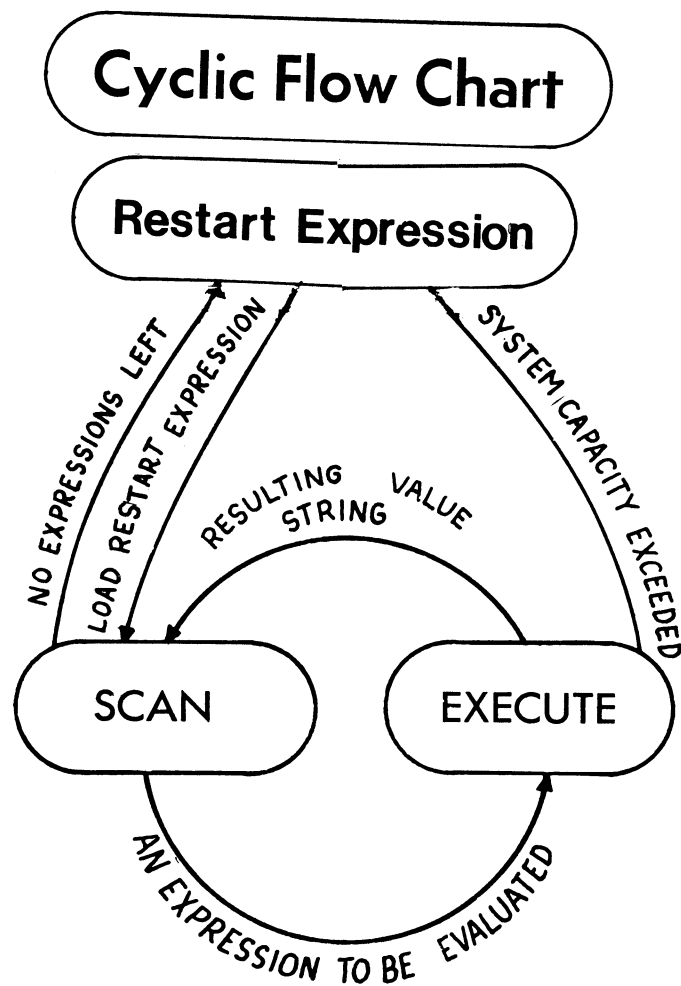
When this point is reached, the computer processor for the SAM76 language automatically reloads the Restart Expression, %OS,%IS//, and begins to scan it. To help you understand there is somewhere near this paragraph a diagram to help you understand the whole process.

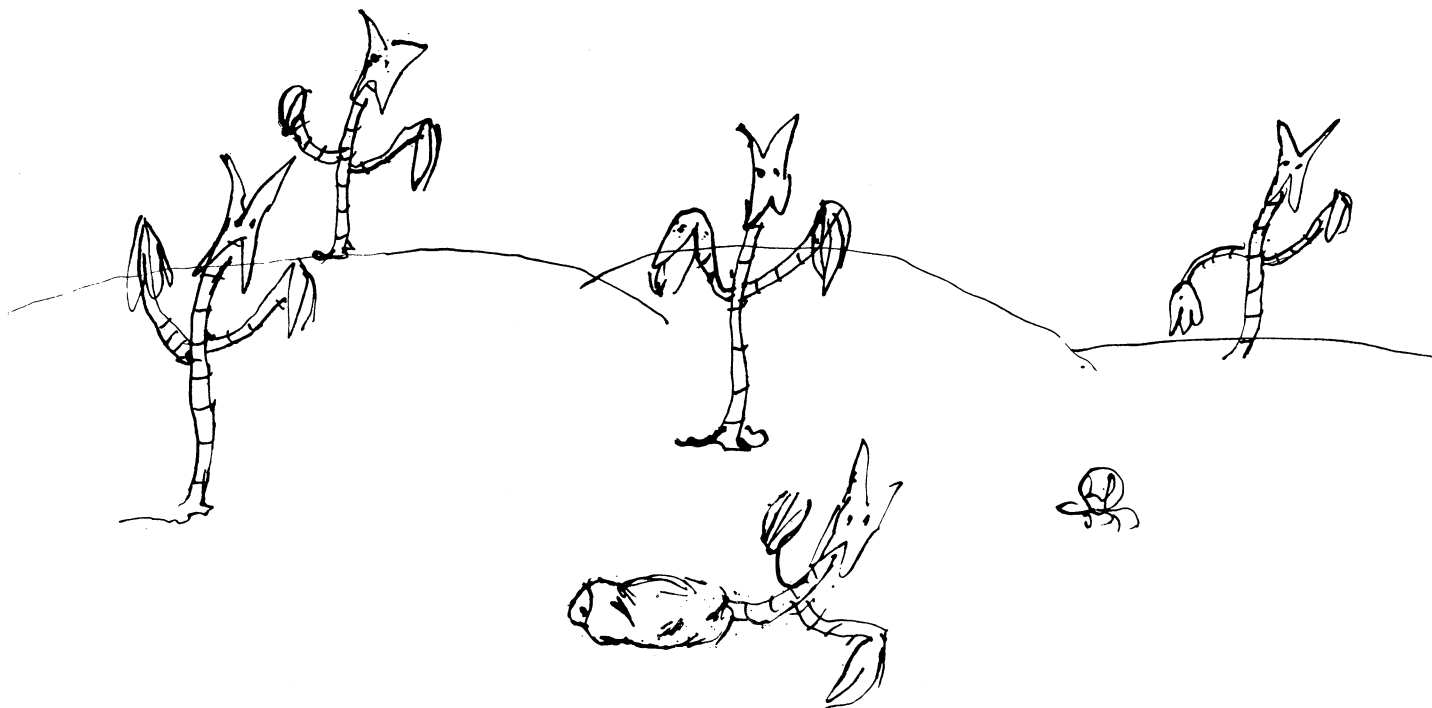
The Restart Expression is of the utmost importance to the language. It makes possible the input and output of strings of unlimited length, while leaving a great deal of flexibility.



You've just absorbed a great deal of information. If you expect to retain it you ought to take a break about now. Have some coffee, take a nap, watch some television,

or find out how easy it is to confuse someone by explaining what you learned to your best friend.





LESSON FOUR - Defining Texts

As the SAM76 language is a text-handling language, and as text is usually stored in memory, the SAM76 language provides text storage capabilities.

The basic storing primitive is Define Text. The two-letter mnemonic for Define Text is DT.

DT Define Text

```
{ }-----  
{ } %DT,A,PEOPLE/=  
{ }  
-----
```

This is an example of a DT primitive expression.

In the Define Text primitive, as in all the SAM76 language primitives, the opening syntax is percent `|%` or ampersand `&|` signs.

Directly following the percent `|%` sign is the first argument, the two-letter mnemonic DT.

The second argument contains the name string (a null string, a character, or a set of characters), by means of which the text string in the third argument can be referenced.

In this case A is the referencing string (name) of the text string PEOPLE. The stored text string, together with its referencing name, is called a "text".

Execution of a Define Text primitive creates a "text". It is the way text is stored in memory.

The slant `|/|` sign and the equal sign have their usual meanings.

The name and text can be no character, any character, or any sequence of characters of any length. But for some characters like the percent `|%` and slant `|/|` signs, the activating character, and in general the various other warning characters, some different techniques must be used when typing them in.

```

{}-----
{}  %DT,3AF DOG ?"T***,**R MT/=
{}
{}-----

```

This is an example of an expression that would not ordinarily have been defined, since its name and text string are made of garbage characters. It shows, we hope, that there is no reason why a "text" (a "text" is a string in storage that can be referenced by a name) cannot contain a name and text of unusual characters.

In the example, 3AF DOG ?"T*** is the name of the text string **R MT.

```

{}-----
{}  %DT,,THE DOG/=
{}
{}-----

```

This is another example. This expression causes the text string THE DOG to be referenced by the null string as the name; in case you didn't notice there are two commas side by side between DT and THE in the example - between them is a null string.

One other characteristic of the DT primitive when used with the |%| is that if there is a "text" in memory with a specific name, and that same name is used as the referencing name of a new "text", the text of the original "text" is erased.

```

{}-----
{}  %DT,A,APPLE/=
{}
{}-----

```

This is an example of what was just said. Previously, the text PEOPLE had the name A. If a new "text" A is defined with text APPLE, A would thereafter reference only the text string APPLE. The "text" in which A referenced PEOPLE is erased. In this way the SAM76 language makes certain that there is only one "text" with any specific name.

There are of course ways of circumventing this feature because no one is ever wise enough to decide for sure what you the user want to do - using the |&| instead of the |%| in front of the DT will cause the addition in the "text" area of the new "text" without removing any existing ones of the same name.

LESSON FIVE - Fetching Texts

In the preceding chapter, the Define Text primitive was discussed. The DT primitive stores text strings under a name by which they can be referenced; therefore the SAM76 language must have some primitives which reference stored "texts".

The basic command for referencing "texts" stored in memory is the Fetch Text primitive:

`%FT,A/|`

FT
Fetch Text

The basic Fetch Text primitive contains two arguments.

The first argument, as in most SAM76 expressions, is the two or three letter mnemonic of the primitive. In this case it is FT. The next argument is the name of the "text" that you want to reference. (Note: since every "text" fetched must have been defined at some earlier time, the "texts" in this book are often shown defined just before the example of the Fetch Text expression itself).

```
{ }-----
{ } %DT,A,APPLE/=
{ }
-----
```

The "text" named A is defined.

```
{ }-----
{ } %FT,A/={APPLE}
{ }
-----
```

The expression `%FT,A/` is scanned, and when the slant `|/|` sign is found, the scanner stops and evaluates the expression.

The value of `%FT,A/` is APPLE since the value of a Fetch Text is the text string of the "text" whose name A is shown in the second argument of the Fetch Text command.

The FT primitive does not have the ability to print out strings; APPLE is printed out in the following manner.

```
|%FT,A/|
```

When this expression is typed in, the scanner really sees %OS,%FT,A/, because the Fetch Text expression automatically replaces the %IS/ in the idling procedure.

The Fetch Text expression is then scanned and replaced by its value which is APPLE. The scanner now sees %OS,APPLE/, which is executed, and APPLE is printed out.

These are examples of Fetch Texts for "texts" that were defined in the last lesson. In the first example, the "text" named 3AF DOG ?"T***/= {**R MT} is fetched, and in the second, the "text" whose name is the null string is fetched and replaced by its value which is THE DOG.

```
{ }-----  
{ } %FT,3AF DOG ?"T***/= {**R MT}  
{ }-----
```

```
{ }-----  
{ } %FT,/={THE DOG}  
{ }-----
```



LESSON SIX - Protection

In the SAM76 language there are some characters which are treated in a different manner from most of the others.

One of the characters that has extra significance in the language is the comma.

An unprotected comma is a delimiter which separates arguments in the SAM76 language.

If one wanted to type out:

{HI, HOW ARE YOU?}

one might try the expression

|%OS,HI, HOW ARE YOU?/|

however, this approach will give the result

{HI}

instead of the desired result

{HI, HOW ARE YOU?}

Only HI was printed out because the comma marked the end of the second argument, and the OS primitive types out only the contents of the second argument. Therefore only HI (the second argument) was printed out.

To print out:

{HI, HOW ARE YOU?}

it is necessary to have the comma protected so that it is seen as a comma and not as a delimiter.

Protection means that the scanner will not see that which is protected. The scanner knows that it is to overlook all characters that may be found between certain pairs of characters, called "Protecting Character Pairs". That means that when the scanner finds one of these characters it immediately zips along looking only for the mate to the first one it found. Then the scanner resumes normal search. Note that the pair that enclosed a "Protected zone" is consumed in this process. For the time being we will only consider the following specific Protecting Character Pair:

Protecting Character Pairs

```
!! ..... /|
```

The protected zone is that number of characters found between the exclamation `!!` point and the slant `|/` sign in the above example.

The protection of a comma can be done in any of the following ways:

```
|%OS,HI!./ HOW ARE YOU?/|
```

would give {HI, HOW ARE YOU?}

```
so would |%OS,H!I./ HOW ARE YOU?/|
```

```
and again would |%OS,!HI, HOW ARE YOU?//|
```

or any other way in which a protecting character pair flanks or surrounds the comma. In all of the preceding examples, the protecting character pairs were not printed out. This is because they were looked at as protective character pairs and not as a plain ordinary pair of protecting character pairs. Strangely enough, for a pair of character pairs to be printed out, they must themselves be protected by another pair of protecting character pairs.

For example:

```
|%OS,SAM76 (A LANGUAGE)/|
```

would result in {SAM76 A LANGUAGE}

```
being printed out, while |%OS,SAM76 !(A LANGUAGE)//|
```

```
will print out {SAM76 (A LANGUAGE)}
```

The same text will be the result if

```
{ }-----
{ } %OS,!SAM76 (A LANGUAGE)//={SAM76 (A LANGUAGE)}
{ }-----
```

is done. The character pairs around A LANGUAGE will always be printed out if there are protecting character pairs flanking them somewhere.

In the SAM76 language, there is one other place where protection is useful. This is in a situation in which it is desired to output a string of characters having additional meaning in the language because the string is an executable expression.

```
{}-----
{}  %OS,%AD,1,1/={2}
{}
{}-----
```

In the example, 2 is printed because the computer would evaluate the add and then print out its value.

If you specifically wanted to print out

```
{%AD,1,1/}
```

the command |%OS,!%AD,1,1///|

would have to be executed.

Note that in all of the above only one type of protective character pair was used, namely the pair consisting of the exclamation "!" point and the slant |/| sign. This combination is usually easier to use because if you are careful keeping track of the number of the warning characters which begin an expression and the slant signs which end each of them you can just type a few extra slant signs for good measure before hitting the activating character.

There are times when you might want to use the "!" and |/| in text without being compelled to keep track of their number, or you may want to use something more distinctive to outline a protected zone. SAM76 allows you to do this by saying that you can also use two other pairs of symbols for protection. Usually the system is initialized with the following pairs assigned for protection.

```
( .... ) and < .... >
```

Interestingly these can be used as shown above where they serve to protect the characters denoted by the row of periods or they can also be used in a reverse mode thus:

```
) .... ( and > .... <
```

Needless to say, but we will anyway, the round parentheses can be protected within the angle brackets and the angle brackets can be protected within the round parentheses; in the protected zone you do not have to have an equal number of begin and end characters other than those which actually start and end the full protected zone.

The above may sound confusing; try it and like it!

Many times you want to have a protected zone which has only one character in it - perhaps a warning character; to save you from having to keep track of the number of begin and end protecting characters and to reduce the amount of typing you have to do, the SAM76 language provides a particular warning character, initially the commercial |@| at to tell the scanner that the immediately following character is to be considered fully protected.

In the example below the two expressions look the same to the scanner:

```
|try it and like it<!>|
```

is the same as:

```
|try it and like it@!|
```

In this way SAM76 language expressions can be handled as easily as ordinary text.



LESSON SEVEN - Procedures

This lesson deals with a use of protection which allows a technique known as procedure writing.

To understand how procedure writing is useful consider the problem of printing out a specific message or paragraph several times. To use Output String command several times would be rather foolish as it would take less time to type the message by hand.

A method which solves the problem with relative ease would be to define a string as an Output String expression which would output the desired message. This expression must be protected otherwise it would be executed before the Define Text expression was stored, causing the message to be printed out and the string defined the null string. If the message were MESSAGE, and the name of the "text" was A, the Define Text expression would look like:

```
{ }-----  
{ } %DT,A,!%OS,MESSAGE///=  
{ }-----
```

The message in this case is quite trivial, but it could be long. At this point it is very important to realize that string A consists of

```
|%OS,MESSAGE/|
```

and not

```
|!%OS,MESSAGE//|
```

The protecting character pair which insured that the expression would not be executed was stripped away when the Define Text expression was stored.

Now if %FT,A/ is executed, MESSAGE will be printed out. This may seem odd as string A did not consist of MESSAGE, but contained the expression %OS,MESSAGE/. The important factor was that A contained an executable expression so when A was fetched the expression

```
|%OS,MESSAGE/|
```

was placed in the working area, not the string:

```
|!%OS,MESSAGE//|
```

Since this expression was executable when it was scanned it was recognized and executed causing the printing of MESSAGE.

By being able to fetch string A over and over again it is possible to have the message typed over and over again. The problem of having the text MESSAGE typed out several times is solved.

A procedure is a "text" which contains one or more executable expressions so that when it is fetched these expressions are scanned and evaluated. The "text" A was a procedure. Another procedure is defined below.

```
{ }-----
{ } %DT,A,!%DT,TEXT,%IS//%OS,
{ } %FT,TEXT/
{ } %FT,TEXT///=
{ }
{ }-----
```

This expression defines a "text" named A as several executable expressions, the whole lot of which are protected to insure that they are not executed when A is defined. This string A is a procedure which when fetched will read in a string terminated by the activating character and will print it out twice. To understand how this procedure works, let us examine "text" A.

```
|%DT,TEXT,%IS//%OS,
  %FT,TEXT/
  %FT,TEXT//|
```

When A is fetched, the first expression executed would be the Input String expression nested within the Define Text expression.

If one had typed:

```
|THIS WILL BE COPIED=|
```

after A had been fetched, it would replace the Input String expression with THIS WILL BE COPIED so that the scanner would see:

```
|%DT,TEXT,THIS WILL BE COPIED/%OS,  
%FT,TEXT/  
%FT,TEXT//|
```

Then a "text" named TEXT is defined as THIS WILL BE COPIED. After this, TEXT is fetched twice, separated by a new-line code. The resulting expression would look like this:

```
|%OS,  
THIS WILL BE COPIED  
THIS WILL BE COPIED/|
```

When this expression is evaluated,

```
{THIS WILL BE COPIED  
THIS WILL BE COPIED}
```

is printed out. Of course all of these procedures occur within the computer and all that the user sees is:

```
{ }-----  
{ } %FT,A/=THIS WILL BE COPIED=  
{ } {THIS WILL BE COPIED  
{ } THIS WILL BE COPIED}  
{ }  
-----
```

Furthermore, this same procedure, A, could be used over and over with different messages each time. Procedures, written in much the same manner although using different combinations of primitives, represent the way programming is done in the SAM76 language. Programming is usually a difficult art performed by trained programmers (or trained monkeys, giraffes,.....) but procedures can be written in the SAM76 language by people who are not necessarily programmers. To differentiate between the two skills, sets of SAM76 language procedures which carry out some function are referred to as scripts instead of programs.

In conclusion, procedures are "texts" which contain executable expressions, which when fetched, are executed. Scripts are combinations of one or more procedures which work together in accomplishing some task. This technique, called procedure writing, is the most powerful and most heavily used feature of the SAM76 language.



LESSON EIGHT - FT Revisited

In the last lesson it was shown that text with executable expressions can be defined in a string by protecting it so that when the "text" is fetched it will be executed. This mechanism of the SAM76 language allows the execution of procedures but makes it very difficult to examine the contents of a "text" which contains such strings. The only way that executable text could be fetched without its being executed would be to have it protected twice. If string A was defined like

```
| %DT, A, !! %OS, MESSAGE///// |
```

string A would contain

```
| ! %OS, MESSAGE// |
```

because the protecting character pair would be stripped away. If string A is fetched, it will be scanned again and the second set of two protecting characters would be removed, but they would protect the active text, and the fetch would result in

```
{ %OS, MESSAGE/ }
```

This method allows one to fetch the string without executing it, but it is now almost impossible to execute the resulting text.

Since it is sometimes necessary to examine the complete unevaluated contents of a procedure there must be, and of course there is, a better and simpler way to output such text without executing it.

The whole problem lies in the fact that the string is scanned when it replaces the Fetch.

A solution to the problem then would be to have a fetch in which the resulting string would not be rescanned. This technique allows easy output of either evaluated or unevaluated text.

This function is not done with a different primitive (a new mnemonic), but by using a different warning character for the fetch primitive so that a fetch to NAME would look like

```
|&FT,NAME/|
```

If NAME were defined instead in the following manner:

```
{}-----
{}  %DT,NAME,!%OS,MESSAGE///=
{}
{}-----
```

a percent |%| sign fetch will execute

```
|%OS,MESSAGE/|
```

causing the printout of

```
{MESSAGE}
```

while an ampersand |&| fetch will return

```
{}-----
{}  &FT,NAME/={%OS,MESSAGE/}
{}
{}-----
```

as the result the fetch was not rescanned. Because scanning does not affect text with no executable expressions, ampersand |&| and percent |%| sign fetches will return the same result when fetching a "text" which contains only such text.

The ampersand |&| technique can also be used with several other primitives which are replaced by value strings which may contain executable expressions.

There is another technique used with the fetch primitive which involves a special kind of fetch. In this fetch the argument containing the FT is left out. As the mnemonic is not there, the action of the primitive FT is stipulated to be implied and so it is called an Implied Fetch.

An Implied Fetch to NAME would look like:

```
|%NAME/|
```

This example would act exactly the same way as a normal percent sign fetch to string NAME:

```
{}-----
{}  %FT,NAME/={MESSAGE
{}  %NAME/={MESSAGE}
{}
{}-----
```

Consider the following procedure:

```
{}-----
{}  %DT,ADD,!%OS,
{}  %AD,%IS/,%IS/////
{}
{}-----
```

If an ampersand `&|` fetch to ADD were executed, it would cause the value of the "text" named ADD to be printed out. The string which would be typed out is:

```
{%OS,
%AD,%IS/,%IS///}
```

When "text" ADD is fetched with a percent `%|` sign so that the expressions are evaluated, two strings hopefully made up of numerals, each terminated by the activating character, will be read in. The strings then will be added together, and then their sum will be printed on the next line as is shown below:

```
{}-----
{}  %FT,ADD/=5=5=
{}  {10}
{}
{}-----
```

An implied fetch could also be used.

```
{}-----
{}  %ADD/=5=5=
{}  {10}
{}
{}-----
```

When implied fetch is used it appears that a new function which causes two numbers to be inputted and their sum printed out has been created with the three-letter mnemonic, ADD.

Implied fetch, because it is inherently useful for handling procedures, is one of the exceptions to the ampersand `&|` syntax rule. An ampersand `&|` implied fetch works exactly the same as a percent `%|` sign implied fetch, and a percent `%|` sign standard fetch: it executes any text. Thus with the "text" ADD:

```
{}-----
{}  %FT,ADD/=27=18=
{}  {45}
{}
{}-----
```

They are all active.

This is desirable as it makes it possible for User Defined Functions to react to percent `|%` sign and ampersand `&` implied fetches in the same way as standard primitives.*

In conclusion, an implied fetch - a fetch primitive with the FT argument implied instead of stated - is useful in dealing with procedures (especially procedures which are artificial primitives). More sophisticated techniques which make possible the writing of user defined primitives like the following will be discussed later on in the book.

```
{ }
{ }  %ADD/=27=18=
{ }  {4 5}
{ }
```

Note *

For an example of how this can be done see the User Defined Functions Section.

Take Ten!

It's break time. Have a sandwich and walk in the sun. Do not proceed without a rest, for it may turn your brain into oatmeal.



LESSON NINE - Listing Text names

You have learned to define "texts" in memory. The place in memory where "texts" are stored is called the "text" area. "Texts" may be defined over long periods of time, and by many different people. Because of this, when someone uses the computer, he may not know the names of all the "texts" in memory.

Since it is sometimes very useful to show the names of all the "texts" in memory, there should be a primitive that does this. In the SAM76 language the primitive that does this is LT.

LT
List Texts

```
|%LT,X/|
```

The value that is returned when a List Texts command LT is executed is a list of the names of "texts" in memory, each of which is preceded by a string, which is designated in the second argument of the primitive expression.

```
{ }-----
{ } %DT,+,RUN/=
{ } %DT,XXX,!%OS,HELLO///=
{ } %DT,HI THERE,1500/=
{ } %DT,?,QUESTION/=
{ } %DT,NAME,?2* 345/=
{ }-----
```

These five "texts" have just been defined.

The following is a List Texts expression with ** in the second argument:

```
{ }-----
{ } %LT,**/={***XX**HI THERE**?**NAME}
{ }-----
```

If the second argument had been left out in the LT expression to the left, it would have been replaced by:

```
{ }-----
{ } %LT,/= {+XXXHI THERE?NAME}
{ }
{ }-----
```

To show another example of LT, the earlier "texts" are removed and the "texts"

```
{ }-----
{ } %ET,+,XXX,HI THERE,?,NAME/=
{ } %DT,HI,HELLO/=
{ } %DT,!%AD,11,5//,COMPLEX NAME ISNT IT/=
{ } %DT,--,----/=
{ }
{ }-----
```

are defined. An LT would result in

```
{ }-----
{ } %LT,**/={**HI**16**--}
{ }
{ }-----
```

if the second argument of the LT was **.

The discrepancy between the real list and the printed list is caused by the fact that the value which replaces %LT,**/ is **HI**%AD,11,5/**--, but when this string is rescanned the primitive %AD,11,5/ is recognized and evaluated. The value, which is 16, then replaces it and this is what is printed out.

It is necessary to use the ampersand |&| instead of the percent |%| sign in the LT command to avoid this problem when any of the "text" names are executable expressions. For example:

```
{ }-----
{ } &LT,**/={**HI**%AD,11,5/**--}
{ }
{ }-----
```

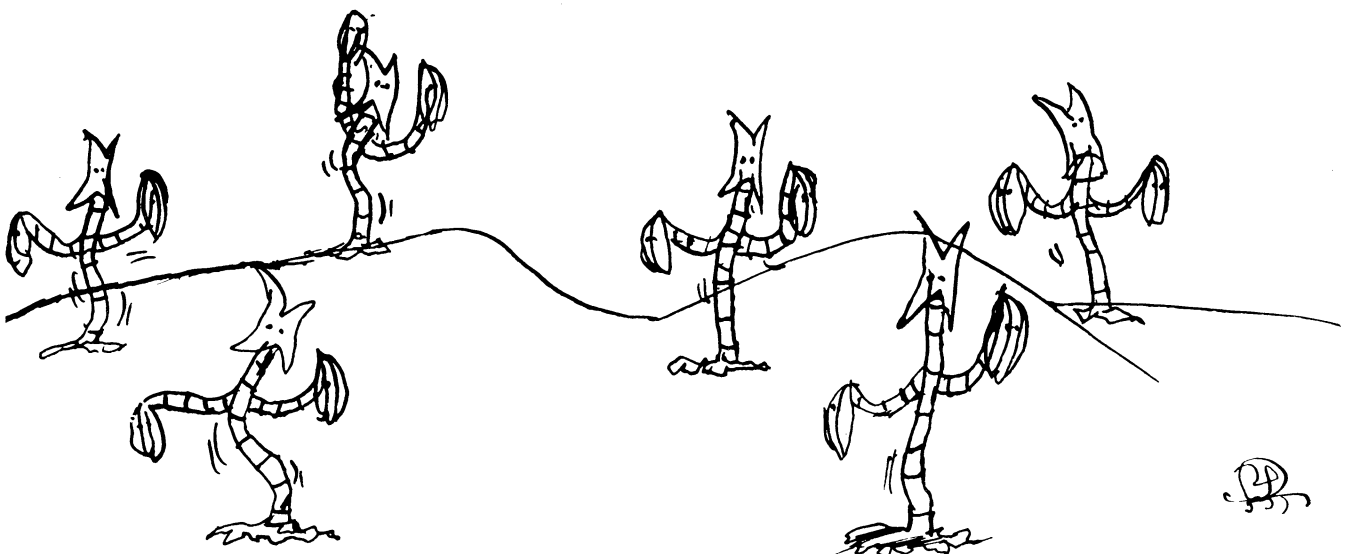
always yields the proper list of the names of the "texts" in memory.

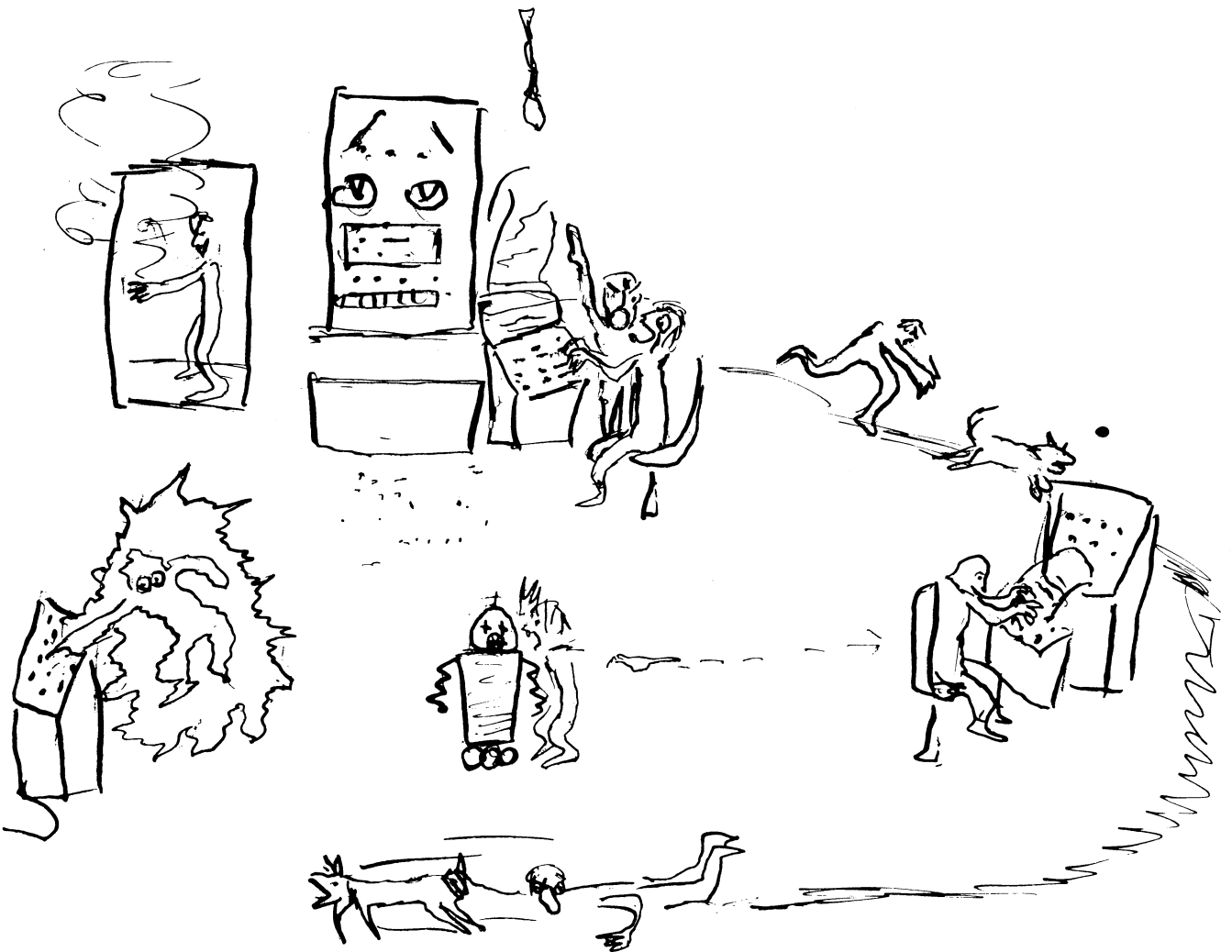
In working with LT primitive, just remember that in order to get a true list, use the ampersand |&|. Also remember that the second argument is a text string which can contain any number or type of characters.

Sometimes an LT command yields nothing. This means that the "text" area or memory is empty.

Note *

The sequence in which the names are listed is earliest defined first using the "S" syntax and last one defined first if the "M" syntax is used.





LESSON TEN - Erasing Texts

When the DT primitive is used, "texts" are left in memory. The number of "texts" that can be defined and stored in memory ("text" area) is limited by the size of the computer. Therefore it is often desirable or even necessary to erase or delete "texts" from the memory.

The SAM76 language contains a primitive called ET (Erase Text) that erases "texts" from memory. For example:

ET Erase Text

```
{}-----
{} DT,A,APPLE/=
{} %DT,B,!%OS,HI///=
{} %DT,C,CAT/=
{}
{}-----
```

In the examples three "texts" were defined with the names A, B, and C. Before the Erase Text command was executed, the value of an <,**/= would have been **A**B**C, but after the command %ET,A,B/= the value of an <,**/= would have only been **C. The command %ET,A,B/= erased the "texts" A and B.

```
{}-----
{} LT,**/={**A**B**C}
{} %ET,A,B/=
{} %LT,**/={**C}
{}
{}-----
```

In the ET command the syntax is the same as usual. The Erase Text command in the preceding example contains three arguments: ET,A,B

The first argument, as is probably obvious by now, is the two-letter mnemonic for Erase Text, ET. The next two arguments are the names of the "texts" to be erased from memory, A and B. The contents of each of these arguments must correspond exactly to the name of the "text" you want to erase.

In the preceding example, the ET primitive contained the names of two "texts" to be erased. However, any number of "texts" can be named for erasure, using successive arguments in the Erase Text primitive. For example, if memory is empty and four strings are defined with the names NAME1, NAME2, NAME3, and NAME ETC., the command:

| %ET, NAME1, NAME2, NAME3, NAME ETC. / |

would erase all the "texts" in memory. If one or more of the "texts" to be erased are nonexistent in "text" area, no erasure action is required. The computer ignores the extra names instead of vomiting at you.

Sometimes it is very desirable to have memory completely empty. If there are some "texts" in memory with names not known, you can clear memory by executing an LT primitive nested inside an ET primitive.

| %ET%LT, !, ///|

When this nested expression is scanned

|&LT,! ,//|

is replaced with its value, which is a list of the names of the "texts" in memory with a comma preceding each one. In the earlier case, when the "texts" NAME1, NAME2, NAME3, and NAME ETC. were in memory, this would result in the scanner now seeing:

```
|%ET,NAME1,NAME2,NAME3,NAME ETC./|
```

This is executed and memory is cleared.

Remember how if you used the &DT,NAME,TEXT/ syntax with an &| you were allowed to duplicate names of "texts"; now if you use the &| with an Erase Text command instead of a %| sign, you will erase ALL occurrences of "texts" with the specified names.

Even though using the nested expression `%ET%LT!,///` is easy, there is an even easier way to clear memory. This is by doing

| %EA/ |

The Erase All primitive, with the two-letter mnemonic EA, is usually used only with one argument. That argument is EA. The Erase All primitive erases all "texts".

With the EA primitive it is very easy to clear memory of all unknown and unwanted garbage.

LESSON ELEVEN - Testing for Identities
--

Very often in a SAM76 language script, a decision or test has to be made to decide whether the action described will follow one path or another.

The primitive that is most often used to control the course of events is the II primitive.

II
If Identical

This primitive compares the strings contained in its second and third arguments character by character. If they are identical, the entire expression is replaced by argument four. However, if they are not identical, the entire expression will be replaced by argument five.

```

{}-----
{} %II,A,B,TRUE,FALSE/={FALSE}
{}
{}-----

```

This is an example of the II primitive. The scanner reads across and recognizes the complete expression as an identity command. The content of the second argument A is compared to the content of the third argument, which in the above example is B. In this case the strings are not identical so the value of the identity is the fifth argument, which contains the string FALSE.

In the II primitive, the final value of the identity is either argument four or five. Therefore in the preceding example, after it was found that A was not identical to B, the string FALSE replaced the II expression %II,A,B,TRUE,FALSE/. At the end, the string FALSE would then be printed out by the output string of the restart expression.

If

```

{}-----
{} %II,A,A,TRUE,FALSE/={TRUE}
{}
{}-----

```

was typed in, the string TRUE would be the outcome, because A is identical to A; thus the value of this expression is argument four.

Next consider:

```
|%II,%AD,57,73/,127,  
!%OS,THE ANSWER IS 127//,  
!%OS,THE ANSWER ISN'T = TO 127///={THE ANSWER ISN'T = TO 127}
```

In this expression, the value of adding 57 and 73 is compared with 127.

The first primitive the scanner sees is %AD,57,73/. This is evaluated and is replaced by 130.

```
|%II,130,127,  
!%OS,THE ANSWER IS 127//,  
!%OS,THE ANSWER ISN'T = TO 127///|
```

This is what the scanner now sees. Since the two executable expressions %OS,THE ANSWER IS 127/ and %OS,THE ANSWER ISN'T = TO 127/ are protected, they are not evaluated.

A test is now made to determine whether the content of the second argument is equal to the content of the third argument. Since 130 is not equal to 127, the value of the II primitive is the string (the protected string) in the fifth argument. Thus, the primitive is replaced by argument five which is:

```
|%OS,THE ANSWER ISN'T = TO 127/|
```

The scanner now looks at this, finding the OS primitive waiting to be executed. The command is now executed, so THE ANSWER ISN'T = TO 127 is printed out.

The results would have been the same if the fifth argument had been THE ANSWER ISN'T = TO 127 instead of !%OS,THE ANSWER ISN'T = TO 127//, except that THE ANSWER ISN'T = TO 127 would have been printed out by the restart expression OS primitive.

If you have been reading along and been kept wondering about the equal sign in the procedure and do not understand how this is possible don't worry for neither do I. I am only the typist, but the authors say it is quite possible and suggested to me that I tell you to keep on going and eventually you will come to a part of the book where it is explained to you in clear and concise terms.

In all of the preceding examples, the contents of both of the arguments could have been determined before the identity expression was ever written. Therefore the decision could have been made without the use of the identity primitive. This is not always the case. Sometimes the computer must make the decision.

A computer should be used to make a decision only when it is easier for the machine to do it.

In both of the earlier examples it was easier for a person to make the decision than for the computer.

II is most useful in a situation in which the final value of the second or third argument is not known at the time the expression is written.

```
|%II,%IS/,HELLO,HELLO,GOODBYE/|
```

Here is an example of an identity in which the final value of the second argument was not known at the time the command was written.

The second argument is the executable expression %IS/. The computer is now waiting for a string to be typed in. If the string HELLO is typed in, the scanner would see:

```
|%II,HELLO,HELLO,HELLO,GOODBYE/|
```

The second and third arguments are identical. Thus the value of the primitive is the fourth argument. HELLO is the value of the fourth argument, so HELLO is finally printed out.

```
{ }-----
{ } %II,%IS/,HELLO,HELLO,GOODBYE/=HELLO={HELLO}
{ }-----
```

If some other string besides HELLO had been typed in, so that arguments two and three were different, the final value of the identity would have been the fifth argument, which is GOODBYE.

```
{ }-----
{ } %II,%IS/,HELLO,HELLO,GOODBYE/=HI={GOODBYE}
{ }-----
```

By using the II primitive in a manner very similar to the way it was just now used, you can design a protection device.

A protection device is a script or procedure for which it is necessary to know a password (or its equivalent) in order to get through to the script and to allow its action to proceed.

```
{ }-----
{ } %DT,SAFE,!%II,%IS/,KEY,!%OS,<
{ } THAT IS RIGHT. OPEN SESAME!>///,!%OS,<
{ } NO, YOU ARE WRONG!
{ } >/%SAFE/////=
{ }-----
```

This is an example of a protective device using the II primitive; note the use of the <....> protective character pair to protect the || used in the text.

This procedure tests whether the typed in characters are the three characters KEY. If they are, the string THAT IS RIGHT. OPEN SESAME! is printed out. If they aren't, a new line code is printed, followed by NO, YOU ARE WRONG! and then another new line code. Following this, the procedure fetches itself again by the implied fetch %SAFE/.

Step by step, the process goes as follows. First the "text" SAFE is set up in the "text" area. Then it is called into action by %FT,SAFE/=. This brings in:

```
|%II,%IS/,KEY,!%OS,<
  THAT IS RIGHT. OPEN SESAME!>//,!%OS,<
  NO, YOU ARE WRONG!
>/%SAFE///|
```

The scanner starts to read across. When it finds the slant | sign that completes the IS command it stops and executes that command. The machine is now waiting for you to type in a string.

If DOOR= is typed in, the scanner would see:

```
|%II,DOOR,KEY,!%OS,<
  THAT IS RIGHT. OPEN SESAME!>//,!%OS,<
  NO, YOU ARE WRONG!
>/%SAFE///|
```

This is now scanned and the identity primitive is evaluated.

Since DOOR is not a perfect match for KEY, the identity expression is replaced by the fifth argument which is:

```
|%OS,<
  NO, YOU ARE WRONG!
>/%SAFE/|
```

This is now scanned and NO, YOU ARE WRONG! followed by a new-line code is printed out, and a fetch to the original procedure SAFE is again made.

This starts the whole procedure all over again so the scanner, instead of getting out of this procedure, is again looking at:

```
|%II,%IS/,KEY,!%OS,<
  THAT IS RIGHT. OPEN SESAME!>///,!%OS,<
  NO, YOU ARE WRONG!
>/%SAFE///|
```

If KEY was typed in, the identity would be satisfied and the fourth argument would be the final value of the identity. The scanner would see:

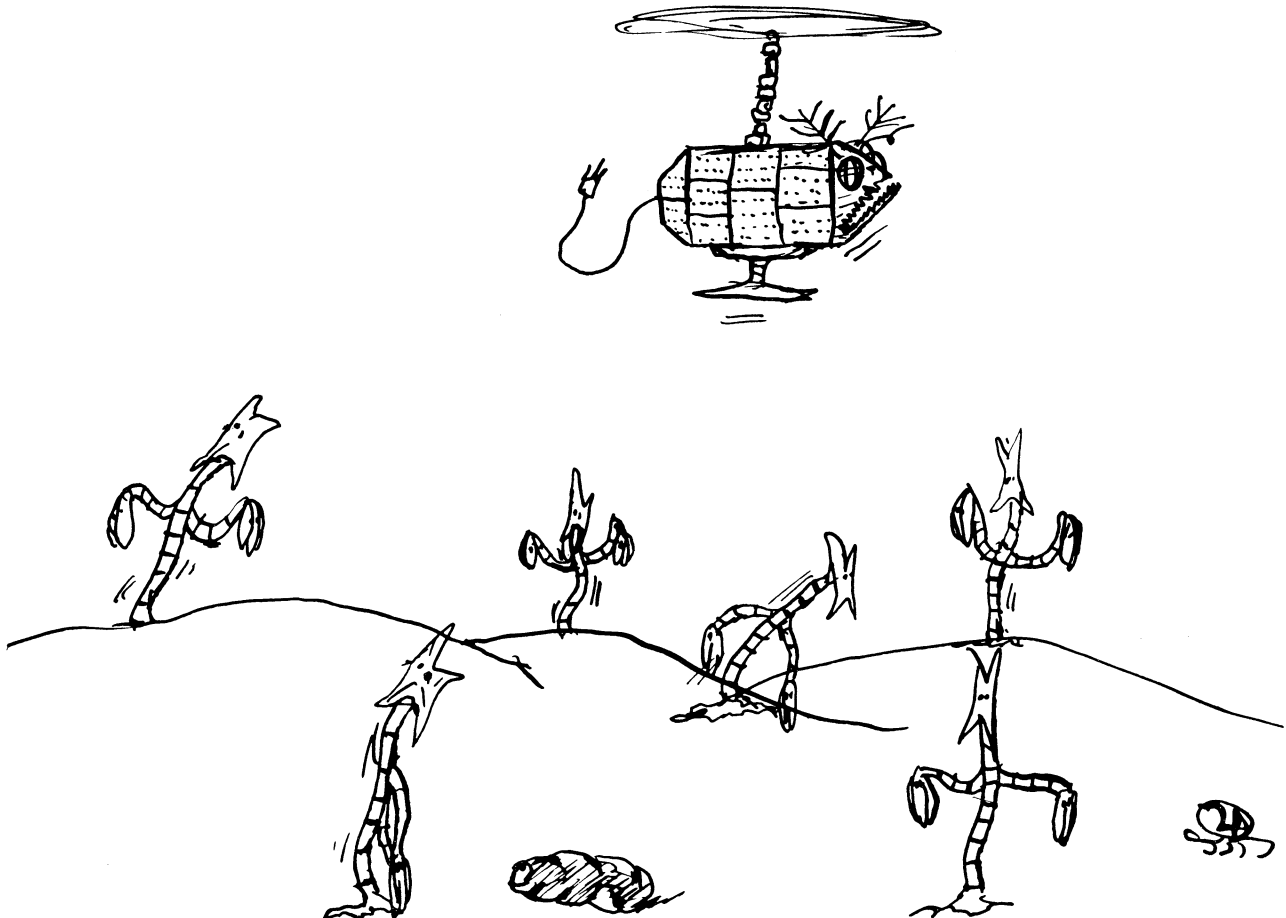
```
|%OS,<
  THAT IS RIGHT. OPEN SESAME!>|
```

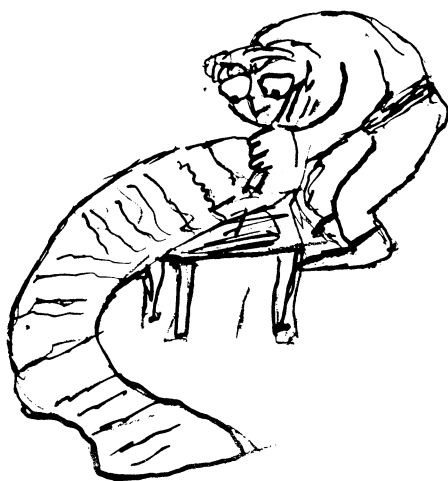
This is now evaluated and

```
{THAT IS RIGHT. OPEN SESAME!}
```

is printed out. The SAFE procedure is not fetched again.

Since the II primitive is most useful when used in conjunction with input primitives, the following lesson will discuss the capabilities of the input primitives of the SAM76 language.





LESSON TWELVE - Interaction

In writing scripts, and in the use of the II primitive, it is extremely useful to be able to type in characters each time you execute a procedure. This manner of working with a script while it is being executed is called interaction. An example of interaction that has appeared earlier is the script:

```
{}-----
{} %DT,SAFE,!%II,%IS/,KEY,!%OS,<
{} THAT IS RIGHT. OPEN SESAME!>//,!%OS,<
{} NO, YOU ARE WRONG!
{} >/%SAFE/////=
{}
{}-----
```

The interaction of this script occurs in the testing process where the typed-in string is compared with KEY.

The IS, Input String, primitive makes it possible for interaction to occur. The usefulness of interaction cannot be overemphasized. Almost anytime a useful script is written there is some section of it that interacts with the user.

Since interaction is so important, it is important to learn all about the IS primitive, which is the major input primitive. An example of an IS primitive being used is:

```
{}-----
{} %AD,2,%IS//=4={6}
{}
{}-----
```

In this example, the IS primitive is the first complete primitive expression, so the scanner stops and waits for the value of the Input String to be typed in. In this case a 4 followed by an equal sign was typed, so the value of the %IS/ is the digit 4. The scanner now sees

```
|%AD,2,4/|
```

It is scanned and evaluated and replaced with its value (the answer is), 6. The digit 6 is now printed out. The example would react in the same manner as %AD,2,4/=6 once the 4 has been typed in.

The IS primitive also works well with the OS primitive as you know from an earlier discussion. An example is:

```
{}-----
{}  %OS,%IS//=HI={HI}
{}
{}-----
```

The IS primitive is evaluated and its value is HI. The scanner now sees %OS,HI/. This is executed with HI being printed out.

Going back to the earlier example %AD,2,%IS//, there is another technique by which 6 can be printed out. An example of this technique is:

```
{}-----
{}  %DT,NUMBER,4/=
{}  %AD,2,%IS//=%FT,NUMBER/={6}
{}
{}-----
```

In this example the IS primitive is executed and replaced by the typed in string %FT,NUMBER/.

```
|%AD,2,%FT,NUMBER//|
```

This is what the scanner now sees. This is now scanned and the expression %FT,NUMBER/ is recognized and evaluated. The fetch primitive is then replaced with the text string value of form NUMBER, which is 4, so at this point %AD,2,4/ is being scanned. This is now executed and the answer, 6, is printed.

For an example of where IS can be a problem, look at the much used example SAFE.

```
{}-----
{}  &FT,SAFE/={%II,%IS/,KEY,!%OS,<
{}  THAT IS RIGHT. OPEN SESAME!>//,!%OS,<
{}  NO, YOU ARE WRONG!
{}  >/%SAFE///}
{}
{}-----
```

In this example, what is really being compared to KEY is not the typed-in string, but the final value of the expression that was typed in.

If KEY is typed in, the final and initial value of KEY is KEY, so the match is made.

But consider the example where %FT,COMBINATION/= is typed in:

```
{}-----
{} %DT,COMBINATION,KEY/=
{} %SAFE/= %FT,COMBINATION/=
{} {THAT IS RIGHT. OPEN SESAME!}
{}
{}-----
```

This occurs even though it is obvious that the string %FT,COMBINATION/ is not identical to KEY. However, the string value, resulting from its evaluation, is identical to KEY. A step by step analysis explains what happens.

When %SAFE/= is typed, the stored procedure is brought forth and the scanner sees:

```
|%II,%IS/,KEY,!%OS,<
  THAT IS RIGHT. OPEN SESAME!>///,!%OS,<
  NO, YOU ARE WRONG!
  >/%SAFE///|
```

This is scanned, and IS is the first executable primitive. The string %FT,COMBINATION/= is then typed in, followed by an equal sign. This string now replaces %IS/, so what is being scanned is:

```
|%II,%FT,COMBINATION/,KEY,!%OS,<
  THAT IS RIGHT. OPEN SESAME!>///,!%OS,<
  NO, YOU ARE WRONG!
  >/%SAFE///|
```

This is now scanned and the first executable primitive is %FT,COMBINATION/. This is now evaluated and replaced by KEY, which is its value. We now have:

```
|%II,KEY,KEY,!%OS,<
  THAT IS RIGHT. OPEN SESAME!>///,!%OS,<
  NO, YOU ARE WRONG!
  >/%SAFE///|
```

This expression is again scanned and the only executable primitive at this point is the II itself. The fourth and fifth arguments, both of which are executable expressions, are protected by a protective character pair. Therefore the II primitive is executed. A test is made to determine whether the second and third arguments are alike. In this case KEY is the same as KEY, so the fourth argument is the value of the identity. This value is %OS,<THAT IS RIGHT. OPEN SESAME!>/. This replaces the II, so %OS,<THAT IS RIGHT. OPEN SESAME!>/ is now scanned.

This is executed, and

```
{THAT IS RIGHT. OPEN SESAME!}
```

is printed.

Hey! Wait a minute! %FT,COMBINATION/ isn't the same as KEY. There's a flaw in the program. The final value of the input string isn't what was supposed to be compared to KEY. The typed-in string itself was to be compared. To solve this problem it is necessary to backtrack and discuss the difference between |%| and |&| fetches.

```

{}-----
{} %DT,MARK,!%OS,TYPE NAMETHEN AN EQUAL SIGN///=
{} %FT,MARK/={TYPE NAMETHEN AN EQUAL SIGN}
{} &FT,MARK/={%OS,TYPE NAMETHEN AN EQUAL SIGN/}
{}
{}-----

```

The difference between a |%| and an |&| fetch is that on an |&| fetch the expression is not rescanned and evaluated while on the |%| fetch it is. In the case of the earlier II

```

| %II,%IS/,KEY,!%OS,<
  THAT IS RIGHT. OPEN SESAME!>//,!%OS,<
  NO, YOU ARE WRONG!
  >/%SAFE///|

```

we do not want the typed-in string to be evaluated. Instead, we want it to be input and then left as it is, so it can be compared with KEY. We do not want the value string produced by the IS to be evaluated.

When it is desired that the value of an IS be left unevaluated, an ampersand is used with the Input String &IS/.

In the example, the procedure should have been written:

```

| %II,&IS/,KEY,!%OS,<
  THAT IS RIGHT. OPEN SESAME!>//,!%OS,<
  NO, YOU ARE WRONG!
  >/%SAFE///|

```

Now the evaluation process is suspended after the value of IS has been found. For example:

```

{}-----
{} %DT,SAFE 2,!%II,&IS/,KEY,!%OS,<
{} THAT IS RIGHT. OPEN SESAME!>//,!%OS,<
{} NO, YOU ARE WRONG!
{} >/%SAFE 2/////=
{}
{}-----

```


Now observe what happens:

```

{}-----
{}  %SAFE 2/=%FT,COMBINATION/=
{}  {
{}  NO, YOU ARE WRONG!
{}  }
{}
{}-----

```

What occurs when SAFE 2 is fetched is that %SAFE 2/ is replaced by:

```

| %II, &IS/, KEY, !%OS, <
  THAT IS RIGHT. OPEN SESAME!> //, !%OS, <
  NO, YOU ARE WRONG!
  > /%SAFE 2///|

```

This is scanned and &IS/ is the first executable primitive. The string %FT,COMBINATION/= is typed in and it now becomes the value of the IS. The scanner is now looking at

```

| %II, %FT, COMBINATION/, KEY, !%OS, <
  THAT IS RIGHT. OPEN SESAME!> //, !%OS, <
  NO, YOU ARE WRONG!
  > /%SAFE 2///|

```

but the expression %FT,COMBINATION/ isn't evaluated because the Input String expression has an ampersand |&|. The identity is now executed and %FT,COMBINATION/ and KEY are not identical, so the fifth argument, which is

```

| %OS, <
  NO, YOU ARE WRONG!
  > /%SAFE 2/|

```

replaces the II expression. The argument is now executed so

```

{
  NO, YOU ARE WRONG!
}

```

is printed out and SAFE 2 is again executed so it is again waiting for a new string to be typed in.

Another example which demonstrates the difference between |%| and |&| fetch is %OS,%IS// and %OS,&IS//.

When

```

{}-----
{}  %OS, %IS//=%AD, 2, 3/={5}
{}
{}-----

```

is done the result will be 5 because the %IS/ is replaced by %AD,2,3/ and then %OS,%AD,2,3// is scanned; since the Input String was a percent |%| sign Input String, the command %AD,2,3/ is evaluated and replaced by 5. Then %OS,5/ is evaluated and 5 is printed out.

When you do

```
{}-----  
{ } %OS,&IS//=%AD,2,3/={%AD,2,3/}  
{ }  
{}-----
```

the result is %AD,2,3/, because the IS is replaced by %AD,2,3/. The expression scanned is %OS,%AD,2,3/, but when the scanning process is reinstated on this expression %AD,2,3/ isn't evaluated because of the ampersand |&| of the IS. The Output String primitive is executed and

{%AD,2,3/}

is printed out.



LESSON THIRTEEN - More Interaction

Under some conditions it is useful to be able to input a specific (limited) number of characters and when this number is reached for the computer to react as if the activator had been typed.

The Input Character primitive does this. Each Input Character primitive expression %IC/ inputs one character, and when that character has been input, the scanner restarts. For example:

IC
Input Character

```

{}-----
{} %AD,%IC/,5/=1{6}
{}
{}-----

```

In the example, the scanner reads across and recognizes the %IC/ as the first executable primitive. IC is the mnemonic for Input Character.

The machine is now waiting for one character to be typed in. A single digit 1 is typed, and it replaces %IC/. The scanner is now looking at %AD,1,5/, and this is scanned and evaluated. The 16 following %AD,%IC/,5/= isn't sixteen at all, but is the typed-in 1 and the answer 6.

If more input characters are needed, it is possible to either use a number of IC primitives following one another, or better yet to use the ID Input "D" character primitive. For example:

ID
Input D
Characters

```

{}-----
{} %AD,10,%ID,2//=51{61}
{}
{}-----

```

This is useful when the typed-in string is being compared in some way with another string.

Using the earlier example SAFE, in a slightly different manner, demonstrates this point.

```

{}-----
{} %DT,SAFE 3,!%II,%IC/%IC/%IC/,KEY,!%OS,<
{} THAT IS RIGHT. OPEN SESAME!>//,!%OS,<
{} NO, YOU ARE WRONG!
{} >/%SAFE 3/////=
{}
{}-----

{}-----
{} %FT,SAFE 3/=KEY
{} {THAT IS RIGHT. OPEN SESAME!}
{}
{}-----

```

The scanner replaces the call with the value of SAFE 3. SAFE 3 is now rescanned and the scanner stops at the first slant | / | sign. This slant sign completes the primitive %IC/ so the scanner stops and the machine is now waiting for one character to be typed. In this case K is typed. The scanner now sees %II,K%IC/%IC/,KEY,etc./. This expression is scanned again and the first executable primitive is the second %IC/. E is now typed, so the value of the IC is E. The scanner continues and it now sees:

```

| %II,KE%IC/,KEY,!%OS,<
  THAT IS RIGHT. OPEN SESAME!>//,!%OS,<
  NO, YOU ARE WRONG!
  >/%SAFE 3///|

```

The next IC is handled in the same way, when it is replaced by Y. Now when the procedure

```

| %II,KEY,KEY,!%OS,<
  THAT IS RIGHT. OPEN SESAME!>//,!%OS,<
  NO, YOU ARE WRONG!
  >/%SAFE 3///|

```

is scanned, the first and only complete command is the II command because the commands in arguments four and five are protected so the II command is executed. Arguments two and three are compared and since they both contain KEY, the fourth argument is the value of II, so

```

| %OS,<
  THAT IS RIGHT. OPEN SESAME!> / |

```

is now scanned and executed. Finally

```

{THAT IS RIGHT. OPEN SESAME!}

```

is printed out.

When Input Character primitives are used, it is not necessary to strike the Activating character. In other words, to input KEY with an IS primitive, KEY would have to be typed, followed by the Activating character. However, when the IC primitive is used, no Activator is necessary.

Besides being useful for inputting strings of specific length, IC can be used to bring in characters that are not normally typable.* For example, imagine trying to execute:

```
|%OS,DOG=CANINE/=|
```

The problem is obvious. When the equal "=" sign is typed following the G in DOG=, execution occurs. Since the expression is not yet complete, the expression does not return the desired value. An IC primitive can be used to solve this problem in the following manner:

```
|%OS,DOG%IC/CANINE/=={DOG=CANINE}
```

Upon scanning, the IC primitive is executed first so the system waits for the user to type a character. If an equal "=" sign is typed, that is the character that becomes inserted in the text string. Finally the text string DOG=CANINE is printed out.

Some characters will affect the string once they are typed. An example of this is:

```
|%OS,%IC/AD,3,4//=%{7}
```

This works as if %OS,%AD,3,4// had been executed, and of course the result will be a 7 printed out. In the same way,

```
|%OS,%IC/AD,3,4//=,|
```

would result in the null string or nothing being printed out because it is the same as %OS,,%AD,3,4// and nothing is in the second argument of that expression. To input characters, and to have them treated non-actively, you must use an ampersand |&| IC like the |&| IS.

The ampersand IC is an extremely powerful tool. It can input any character and have it remain inactive because it is not scanned.

The ampersand IS, IC or ID is most useful for strings that involve an unbalanced number of protective character pairs. A string with an unequal number of begin and end protected string warning characters will not execute normally. It will abort (not do anything). But if ampersand IS, IC or ID is used, any number of odd characters from the currently used protective character pair can be used.

In this example:

```
{ } ~~~~~
{ } %DT,SLANT,%IC//=/
{ } %FT,SLANT/=
{ }
~~~~~
```

the result is the null string because the slant `|/|` sign that was typed in was recognized as a warning character, so the scanner was looking at

```
|%DT,SLANT,||
```

and it closed the Define Text expression with the first slant `|/|` sign.

This example

```
{ }-----
{ } %DT,SLANT,&IC//=/
{ } &FT,SLANT/={/}
{ }
{ }-----
```

causes a slant `|/|` sign to be printed out because an ampersand `|&|` Input Character command was used. Note that an ampersand Fetch primitive must be used to fetch such unusual strings.

Note *

It is here that you find out how we got an equal "=" sign in the text string in the preceding lesson.

Hold it!

Consider for a moment that the ability of your brain to absorb information diminishes the greater the amount of information presented at one time. With this fact in

mind determine for yourself whether it might not be better to take a rest now.



LESSON FOURTEEN - Partitioning Texts

By using the techniques so far discussed it is possible to define, erase, and fetch "texts". The techniques that will be demonstrated in this lesson make it possible to modify or change "texts".

The primitive that modifies "texts" is called Partition Text (its mnemonic is PT).

```
| %PT, A, TEXT /= |
```

PT Partition Text

In the example the syntax has the same meaning as in all other SAM76 expressions.

The first argument is PT, the mnemonic symbol for Partition Text. The next argument is A. A is the name of the "text" (string) that is to be modified. The third argument, TEXT, is the string of characters that are to be replaced by a partition, or deleted from, the text string of "text" named A.

```
{ }-----
{ } %DT, A, A MOUSE AND A DOG /=
{ } &FT, A /= { A MOUSE AND A DOG }
{ } %PT, A, AND /=
{ } &FT, A /= { A MOUSE  A DOG }
{ }
{ }-----
```

The Partition Text expression modifies the "text" and causes a subsequent fetch to yield A MOUSE~A DOG. The two spaces surrounding the AND were left intact.

The Partition Text expression has removed the word AND from the "text" A. This removal of AND is the obvious effect of the Partition Text command. There is another effect. In place of the word AND, an invisible marker has been left behind in the "text" that signifies the presence of the spot where one or more characters have been removed. The name for this spot is a "partition space" and the name of the marker is the "partition". This partition does not appear when an ordinary fetch is executed, not even as a space. This was shown when a fetch to "text" A was done earlier. The useful aspect of a partition is not in the partition itself, but in the fact that a character or string of characters may be substituted for the partition.

For example, in the "text" A that was partitioned earlier, a plain fetch resulted in A MOUSE~A DOG, but if a fetch with one extra argument were performed, the contents of that new argument would be substituted for the partition.

```
{}-----
{}  &FT,A,OR/={A MOUSE OR A DOG}
{}
{}-----
```

So if A was fetched with OR in the third argument, the result would be A MOUSE OR A DOG.

Thus the "text" A was fetched with OR substituted for the partition. Notice that the original word which was deleted had three characters, while the word that replaced it had two. This was possible because any number of characters, greater or smaller than the number of characters partitioned out, may be inserted into a partition space.

Even though A was fetched with OR substituted for the partition, it is important to realize that the "text" A was not affected. The only effect of the fetch was that A was printed out with a specific string of characters, OR, in the partition space.

To make a permanent change in a "text" it is necessary to go through a two step process. First, it is necessary to partition the "text". Since "text" A was partitioned earlier, it is possible to define a new "text" B without again partitioning the "text" A. "Text" B is going to be defined as "text" A, with OR substituted for the partition left by the removal of AND. We do this by:

```
{}-----
{}  %DT,B,%FT,A,OR//=
{}
{}-----
```

This works because the value of %FT,A,OR/ is A MOUSE OR A DOG.

The scanner now sees

```
|%DT,B,A MOUSE OR A DOG/|
```

which is performed, and "text" B is defined as "text" A with OR inserted in place of AND.

The same technique can also be used to redefine "text" A. The original A is still partitioned on AND, so it is possible to redefine "text" A as a fetch to itself with OR in the partition .

```
{}-----
{}  %DT,A,%FT,A,OR//=
{}
{}-----
```


A is now defined as:

```
|A MOUSE OR A DOG|
```

The process of defining the new "text" A causes the erasure of the former "text" A. Therefore "text" A has been totally redefined with the new value. Another point to be made is that upon redefinition, the partitions cease to exist and the replacing string, if any, is permanent in the new "text".

Text A is now redefined and "text" A is permanently changed. The value of a fetch to "text" A now yields A MOUSE OR A DOG.

Another capability of Partition Text is its ability to remove every occurrence of a particular combination of characters. When AND was partitioned out of "text" A there was one partition. Had there been two ANDs, there would have been two partitions.

```
{ }-----
{ } %PT,A,A/=
{ } %FT,A/={ MOUSE OR DOG}
{ }
{ }-----
```

When "text" named A was partitioned on the single character A, it was partitioned twice. Partitions are left where each A was removed. Since there were two A's there are two partitions left in the two spots where there were A's.

A fetch to "text" A with an * used to fill the partitions (that were caused by the removal of the A's) causes the * marks to fill the partition spaces in the value string of the fetch.

```
{ }-----
{ } %FT,A,*/={* MOUSE OR * DOG}
{ }
{ }-----
```

To fill in the partition spaces permanently, you must redefine "text" A.

```
{ }-----
{ } %DT,A,&FT,A,THE//=
{ } %FT,A/={THE MOUSE OR THE DOG}
{ }
{ }-----
```

In the example, "text" A has been redefined by a fetch to "text" A with THE plugged into the partition spaces. Now, when "text" A is fetched, the result is THE MOUSE OR THE DOG. The partition spaces have been filled.

Sometimes it is necessary to modify two or more sections in a string as in the example:

```

{}-----
{} %DT,C,THE HOUSE ON THE HILL/=
{} %PT,C,THE/=
{} %DT,C,&FT,C,A//=
{} &FT,C/={A HOUSE ON A HILL}
{} %PT,C,HILL/=
{} %DT,C,&FT,C,ROAD//=
{} &FT,C/={A HOUSE ON A ROAD}
{}
{}-----

```

The problem is to change the value of the string in "text" C from THE HOUSE ON THE HILL to A HOUSE ON THE ROAD. Using the techniques already shown in this lesson, it is possible to achieve this end. It is done by redefining "text" C with the partition spaces filled with A and ROAD. This is a long and involved process. By using another capability of Partition Text, it is possible to greatly shorten this process.

This new capability is the ability of the Partition Text primitive to replace with partitions more than one specific combination of characters at one time. This is done by adding extra arguments to the Partition Text function.

```

{}-----
{} %DT,C,THE HOUSE ON THE HILL/=
{} &FT,C/={THE HOUSE ON THE HILL}
{} %PT,C,THE,HILL/=
{} &FT,C/={ HOUSE ON }
{}
{}-----

```

Every argument of PT after the second argument is a string of characters to be partitioned out of the "text" whose name appears in the second argument.

This is what happens. First you replace with partitions THE and HILL from the "text" named C. So far, the fetch primitive has only been depicted as having at most three arguments. But to complement the ability of Partition Text, fetch has the additional ability to fill in an unlimited number of partition spaces.

```

{}-----
{} %DT,C,&FT,C,A,ROAD//=
{} &FT,C/={A HOUSE ON A ROAD}
{}
{}-----

```

To fill in the partition spaces created by the removal of THE and HILL, "text" C is redefined with a fetch of "text" C with the partitions to be replaced by A and ROAD.

As was said, Partition Text can replace with partitions more than one character combination at one time. To illustrate this, a new symbolism is needed.

Consider that partitions have numbers that differentiate one partition from another. A "number one" partition will be shown as [1], a "number two" partition will be shown as [2], a "number three" partition as [3], and so on.

VT
Viewing Text

Thus if

```
{ }-----
{ } %DT,D,A DOG AND A MOUSE/=
{ } %PT,D,A,MOUSE/=
{ }
```

is executed, "text" D will have "number one" partitions wherever there was an A in the string. There will be a "number two" partition wherever there was a MOUSE in the string. Thus using these symbols, the string in the "text" named A would look like:

```
{ }-----
{ } %VT,D/{[1] DOG [1]ND [1] [2]}
{ }
```

The result of a normal fetch of "text" D would now look like DOG ND.

```
{ }-----
{ } %FT,D,$,+/{ $ DOG $ND $ +}
{ }
```

Here a dollar sign has replaced all the "number one" partitions and a plus sign has replaced all the "number two" partitions.

Notice that the third argument in a Partition Text primitive is for the "number one" partition, the fourth argument is for the "number two" partitions, and so on, in both the Partition Text and the fetch primitives. There can be as many different partitions as are needed. A problem arises from this, though. Suppose in the original "text" A, which was defined as A MOUSE AND A DOG, the Partition Text commands as shown were executed:

```
{ }-----
{ } %DT,A,A MOUSE AND A DOG/=
{ } %PT,A,A/=
{ } %PT,A,O/=
{ }
```

After they are executed, every partition is a "number one" partition, and since both partitions are marked the same, it is impossible to distinguish between them: [1] M[1]USE [1]ND [1] D[1]G.

```
{}-----
{}  &FT,A/={ MUSE ND DG}
{}  &FT,A,$,+/{ $ M$USE $ND $ D$G}
{}
{}-----
```

When you are editing strings, it is important that two Partition Text operations are not executed this way by mistake. If for some strange reason two Partition Text expressions must be used, and the two types of partitions must be kept separate, it is best done in the style shown:

```
{}-----
{}  %DT,A,A MOUSE AND A DOG/=
{}  %PT,A,A/=
{}  %PT,A,,O/=
{}  &FT,A,*,??/={ * M??USE *ND * D??G}
{}
{}-----
```

Now the partitions for O are "number two", and the partitions for A are "number one". Other instances may require more than two types of partitions, but the technique in handling them is the same. The method shown in the example should only be used when more than one PT command must be used. In normal situations, it is much more convenient to combine the two Partition Text operations in one Partition Text expression, as shown:

```
{}-----
{}  %DT,A,A MOUSE AND A DOG/=
{}  %PT,A,A,O/=
{}  &FT,A,*,??/={ * M??USE *ND * D??G}
{}
{}-----
```

A most useful use of the PT primitive is with the ampersand |&| instead of the usual |%| syntax. Using the |&| form as shown in the following example, only the first immediately next available matching substring in the desired "text" will be partitioned out:

PT
ampersand form

```
{}-----
{}  %DT,A,THE DOG AND THE CAT/=
{}  &PT,A,THE,THE/=
{}  &FT,A,HIS,HER/={ HIS DOG AND HER CAT}
{}
{}-----
```

Now that the basics of the PT primitive have been discussed, it is time for demonstration of a few of the techniques that can be used with it. One of these deals with partitioning out executable expressions.

```

{}-----
{}  %DT,ADD,!%AD,3,%MU,4,2///=
{}  &FT,ADD/={%AD,3,%MU,4,2//}
{}  %PT,ADD,!%MU,4,2///=
{}
{}-----

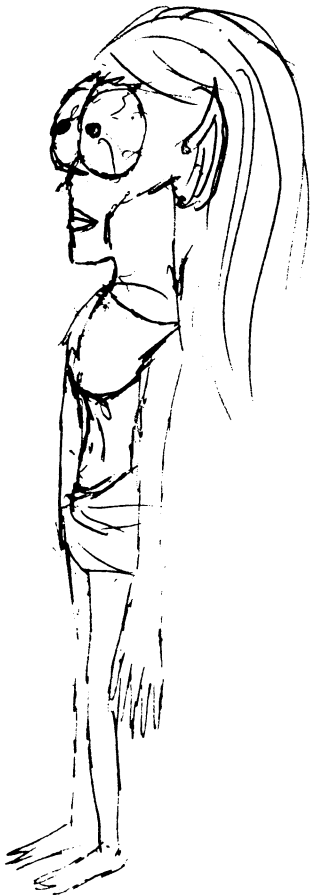
```

The problem is that such strings must be protected. For example, to change the multiply expression %MU,4,2/ to the add expression %AD,6,1/, it is necessary to first replace with partitions the multiply. Notice the multiply expression is protected when being partitioned. If it were not, it would multiply four times two before partitioning. The Partition Text expression would then look like %PT,ADD,8/. The scanner would look for an 8 in the "text" ADD, but there is none. The machine wouldn't blow up just because there is no 8, but the result would not be what you desired.

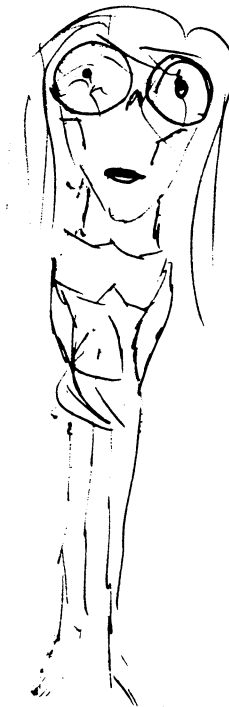
Stop!

If you want to remember what you've read,
you'd better stop reading now!

Fortran Programmers*



* Reference only - too dangerous
to encounter, not to be....





LESSON FIFTEEN - Partitioning Texts - cont'd
--

```

{}-----
{}  &FT,ADD,*/{%AD,3,*}/
{}
{}-----

```

The example is what the partitioned string ADD looks like. The asterisk marks the location of the partition. Using the newer symbolism the "text" ADD looks like

```
|%AD,3,[1]/|
```

To put the command %AD,6,1/ permanently into the partition space, it is again necessary to protect %AD,6,1/ in the fetch expression. This would refill the partition space and the redefined string in "text" ADD is shown by the fetch below.

```

{}-----
{}  %DT,ADD,&FT,ADD,!%AD,6,1////=
{}  &FT,ADD/{%AD,3,%AD,6,1//}
{}
{}-----

```

Another powerful technique involves the ability of the SAM76 language to handle implied fetches along with the Partition Text command. This technique makes possible the writing of scripts with implied fetches that react in very much the same manner as SAM76 language primitives do.

If someone wanted a primitive with the mnemonic ADD that does the same thing that the AD primitive does, it would be possible for him to develop this primitive in a manner similar to the examples.

```

{}-----
{}  %DT,ADD,!%AD,**,*///=
{}  %PT,ADD,**,*/=
{}  &FT,ADD/{%AD,,/}
{}
{}-----

```

At this point the value of ADD is %AD,,/.

What the scanner sees is the value of ADD with partitions marked.

```
|%AD,[1],[2]/|
```

If you wanted to add five and six using the standard AD primitive, you would use an AD with five and six in the last two arguments. The result would be 11.

```
{ }-----
{ }  %AD, 5, 6/={11}
{ }
{ }-----
```

Using the string ADD just defined it is also possible to add five and six.

```
{ }-----
{ }  %ADD, 5, 6/={11}
{ }
{ }-----
```

How this is done is shown in the example using our implied fetch to ADD.

Since the string ADD has two partitions, when five and six are put in the third and fourth arguments of a fetch expression the final result is 11. This occurs because the fetch of the procedure ADD causes six and five to be inserted in the partition spaces.

```
|%AD, 5, 6/|
```

This is what the scanner sees. This is now evaluated, and replaced with its value 11, which is then printed out. The implied fetch, as shown, works in the same manner as the standard fetch.

```
{ }-----
{ }  %FT, ADD, 6, 5/={11}
{ }
{ }-----
```

This is replaced with its value, which is %AD, 6, 5/. This is evaluated and replaced by its value, 11, which is then printed out.

In this way the procedure ADD defined in the SAM76 language is equivalent to the standard primitive AD.

Being able to make copies of SAM76 language primitives is one of the obvious uses of this capability. This ability of the language can be used to develop new useful expressions in the language which are called functions. On the following pages are two examples of such functions. These defined functions almost behave as if they were new primitives.

Here is the development of a defined function named SQ which finds the square of the number in the second argument of the function.

First function SQ is defined as

```
{}-----
{}  %DT,SQ,!%MU,*,*///=
{}  %PT,SQ,*/=
{}  &FT,SQ/={%MU,,/}
{}
{}-----
```

and then the * is partitioned to create two partitions, both "number one":

```
|%MU,[1],[1]/|
```

These are fetches to SQ:

```
{}-----
{}  %FT,SQ,2/={4}
{}  %SQ,2/={4}
{}
{}-----
```

This is what the scanner sees:

```
|%MU,2,2/|
```

It doesn't matter if a standard or implied fetch is given; the result of the fetch will still be 4.

```
{}-----
{}  %SQ,13/={169}
{}
{}-----
```

Evaluation of this example is the same as it was with the evaluation of %SQ,2/, except that the value is 169 and not 4.

Another example of a function defined in the SAM76 language is a function for replacing one string of characters with another string. Follow the examples to see how this is done.

```
{}-----
{}  %DT,REP,!
{}  %PT,***,**/
{}  %DT,***,&FT,***,*//
{}  // =
{}  %PT,REP,***,**,*/=
{}
{}-----
```

The "text" REP in "text" area looks as follows:

```
|%PT,[1],[2]/
%DT,[1],&FT,[1],[3]//|
```

To see how to change BY THE STREAM to ALONG THE ROAD, by means of the defined function REP (Replace), keep watching.

```
{ }-----
{ } %DT,123,THE WAY BY THE STREAM IS LONGER/=
{ }
```

Using REP we can now redefine 123 to be THE WAY ALONG THE ROAD IS LONGER.

```
{ }-----
{ } %REP,123,BY THE STREAM,ALONG THE ROAD/=
{ } &FT,123/={THE WAY ALONG THE ROAD IS LONG}
{ }
```

This happened because %REP,123,BY THE STREAM,ALONG THE ROAD/ is replaced by its value:

```
|%PT,123,BY THE STREAM/
%DT,123,&FT,123,ALONG THE ROAD//|
```

Remember that the partition 1 is replaced by the contents of the third argument of a standard fetch, and the second argument of an implied fetch. The second argument 123 of function REP replaces the partitions numbered 1, the argument BY THE STREAM replaces the partitions numbered 2 and ALONG THE ROAD replaces the partitions numbered 3. The resulting expression is then scanned and evaluated with %PT,123,BY THE STREAM/ being the first executable primitive. This leaves a partition in the "text" named 123. The fetch of "text" 123 with ALONG THE ROAD put in the partition space is the next primitive evaluated, and this is replaced by its value which is THE WAY ALONG THE ROAD IS LONGER. The final expression %DT,123,THE WAY ALONG THE ROAD IS LONGER/ is now executed. BY THE STREAM has been replaced by ALONG THE ROAD in a "text" named 123.

Using this technique, it is possible for the user of the language to define his own set of user functions. These are called "user defined functions."

One more technique should be demonstrated.

When an exclamation || point needs to be partitioned out, it must be handled in one of several possible odd ways.

```
{ }-----
{ } %PT,NAME,!/=
{ }
```

To try this way would be foolish, because the exclamation point will be considered as a match for the slant | / | sign, and not as a character to be partitioned out.

What will work, though, is to use an ampersand | & | input character, as in the example:

```
{}-----
{}  %PT,NAME,&IC//=!
{}
{}-----
```

When the example is executed the computer will be waiting for the user to input a character. This can be an exclamation point which, because the IC has two sharp signs, will not be rescanned. Thus it will be partitioned out of "text" NAME.

Remember that Partition Text finds and removes all occurrences of whatever string of characters is being partitioned out. When you are working with Partition Text, to get at one particular character combination it may be necessary to replace with a partition a longer set of characters so as to make sure that this is the only occurrence of that combination.

An example would be trying to replace with a partition the word A and changing it to THE in a "text" Z defined as A CAT AND A DOG.

```
{}-----
{}  %DT,Z,A CAT AND A DOG/=
{}  %PT,Z,A/=
{}  %FT,Z/={ CT ND DOG}
{}
{}-----
```

After partitioning the "text" on A, a fetch of "text" Z would yield CT ND DOG, which is not the desired result. Therefore it is necessary to replace with a partition the combination A and space (A^):

```
{}-----
{}  %DT,Z,A CAT AND A DOG/=
{}  %PT,Z,A ^=
{}  %FT,Z/={CAT AND DOG}
{}
{}-----
```

This yields CAT AND DOG. You can then put THE into the partition spaces, and redefine "text" Z:

```
{}-----
{}  &FT,Z,THE /= {THE CAT AND THE DOG}
{}  %DT,Z,&FT,Z,THE // =
{}
{}-----
```

Pause for ... !

It is now time for you to either sit down with a hot cup of coffee,* or go to sleep, so as to be fresh for the next lessons.

* The authors, being ignorant in these matters as we have been too busy writing to be drinking (although you may think otherwise), cannot suggest anything more realistic.



```
XDT,A,I AM AN ATTRACTIVE GIRL/=
XPT,A,GIRL,AN,ATTRACTIVE/=
XVT,A/=
I AM [2] [3] [1]
XFT,A,MONSTER,A,HORRIBLE/=I AM A HORRIBLE MONSTER
```



LESSON SIXTEEN - Fetching Elements

The partitions caused by the Partition Text instructions provide the markers for a new type of fetch: the primitive Fetch Element.

```
|%FE,A,Z/|
```

FE Fetch Element

A is the name of the string of the "text" being fetched. The Z is known as a default argument. To understand what this Z argument does, you must first understand what FE does.

If a "text" A is partitioned, it looks like this:

```
{ }-----
{ } %DT,A,HELLO THERE/=
{ } %PT,A, /=
{ }
-----
```

```
|HELLO[1]THERE|
```

If it is fetched with an ordinary fetch primitive, the asterisk replaces the partition.

```
{ }-----
{ } %FT,A,*/= {HELLO*THERE}
{ }
-----
```

The partition divides the string into two parts or elements, HELLO and THERE. If a Fetch Element to "text" A is executed, the result is the first element of the "text" A, which is HELLO.

```
{ }-----
{ } %FE,A/= {HELLO}
{ }
-----
```

If another FE is executed on string A, what is given is the second element, which is THERE.

```

{}-----
{}  %FE,A/={THERE}
{}
{}-----

```

Now there are no more elements in A to fetch, so here is where the default argument comes in. When a Fetch Element is now executed and there are no more elements to fetch, the value produced is the default argument.

This is shown by:

```

{}-----
{}  %FE,A,Z/={Z}
{}
{}-----

```

If the default had been a procedure, the procedure would have been executed.

```

{}-----
{}  %FE,A,!%AD,3,3//={6}
{}
{}-----

```

If the procedure had been a fetch to a string, that string would have been executed.

A curious property arises when you are working with Fetch Element. Suppose we had our original "text" A with the value of HELLO[1]THERE with the partition marked. An FE fetch produces HELLO.

The fact that this result is hardly the whole string of "text" A is due to a device called a "text divider". Every time Fetch Element is executed, this divider moves over to the beginning of the next element to be fetched. That is why the value of each consecutive Fetch Element command is the next element; it just fetches the element between the divider and the next partition. If a normal fetch is done, the result is just the string between the divider and the end of the string. The divider is not moved by the normal fetch. In FE the default argument is only produced when the divider is at the end of the string.

Since it is sometimes useful to return the divider to the beginning of a string there is another primitive named Move Divider which does this.

MD
Move Divider

```
|%MD,A/|
```

The MD primitive is used like this where A again is the name of the "text" containing a string. This command is important when dealing with FE and the other special fetch primitives. These will be discussed shortly.

One further point is worth mentioning before concluding with the FE primitive: the technique of handling elements made up of executable expressions. If there existed a "text" B

```
{}-----
{}  %DT,B,!%AD,1,2/*%OS,HI///=
{}  %PT,B,*/=
{}
{}-----
```

with a value of

```
|%AD,1,2/[1]%OS,HI/|
```

and an FE command was done the result would be three:

```
{}-----
{}  %FE,B,Z/{3}
{}
{}-----
```

The second FE command would have HI printed out.

```
{}-----
{}  %FE,B,Z/{HI}
{}
{}-----
```

This effect is due to the fact that the elements are commands and are executed before they are printed. To print out the unevaluated string, however, it is necessary to use an ampersand Fetch Element.

```
{}-----
{}  %MD,B/=
{}  &FE,B/{%AD,1,2/}
{}  &FE,B/{%OS,HI/}
{}
{}-----
```

Using an ampersand is an easy way to keep from being surprised by funny results. But in truth I usually forget to use it.

Closely related to the FE primitive is the FDE primitive which allows several elements to be fetched at the same time, and that in either a right to left or left to right direction.

FDE Fetch D Elements

The example shown below illustrates the use of FDE on a "text" named A in which the spaces are replaced with partitions. You can look at what was printed out and figure it out, or better yet try it out on your own computer.

```
{ }-----  
{ } %DT,A,ONE TWO THREE FOUR/=   
{ } %PT,A, /=   
{ } %FDE,A,3/={ONETWOTHREE}   
{ } %VT,A/={ONE[1]TWO[1]THREE[1][^]FOUR}   
{ } %FDE,A,-2/={TWOTHREE}   
{ } %VT,A/={ONE[1][^]TWO[1]THREE[1]FOUR}   
{ }-----
```



LESSON SEVENTEEN - Fetching Characters
--

The next two types of fetches are similar to the FE primitive in that they fetch out only part of the string. They differ from the Fetch Element primitives in that they don't deal with partitions, but with the number of characters they fetch out.

The simplest of the two is Fetch Character. It is used in the format:

|%FC,A,Z/|

FC
Fetch Character

where A is the name of a string, and Z is the default argument. The format then is the same as in FE. FC, as the name implies, fetches one character, and in doing so, it moves the divider to the right until it points to the next character. Thus the next fetch character would fetch the next character in the string. An example of how fetch character is used follows:

```

{}-----
{} %DT,A,1234567890/=
{} %FC,A,Z/{1}
{} %FC,A,Z/{2}
{} %MD,A/=
{} %FC,A/{1}
{}
{}-----

```

The primitive FDC for Fetch D characters is a little more complicated.

|FDC,A,D,Z/|

FDC
Fetch D
Characters

It will fetch a specific number of characters, and is used in the format where A is the name of a string, D is the number of characters to be fetched, and Z is the good old default argument.

Using the "text" A, which has been defined as 1234567890, first do %MD,A/=, then %FDC,A,5,2/=. The result of this will be 12345 and the "text" pointer will be left pointing to the six.

Just as in Fetch Character, the divider is left pointing to the next character to be read.

An interesting capability of FDC is that if a negative D is used, then D characters to the left of the divider are read.

To clarify this, suppose a "text" B is created.

```
{}-----
{}  %DT,B,THE HOUSE ON THE HILL/=
{}
{}-----
```

To show it in memory, a symbolic table will be made with an arrow to show the divider position.

The first nine characters (the space is a character) are printed out.

```
{}-----
{}  %FDC,B,9/={THE HOUSE}
{}
{}-----
```

In memory this happened:

```
|THE HOUSE ON THE HILL
|^      ^|
```

The divider moved from the first arrow to the second. A normal fetch will now give:

```
{}-----
{}  %FT,B/={ ON THE HILL}
{}
{}-----
```

These are all the characters to the right of the divider, to the end of the string. However, if a negative FDC is used the result is:

```
{}-----
{}  %FDC,B,-5/={HOUSE}
{}
{}-----
```

What occurred in the computer's memory was that the five characters HOUSE were read, and the divider was moved to the new position.

```
|THE HOUSE ON THE HILL
|  ^|
```

The word HOUSE was returned frontwards (in other words, when the dividers move backwards, they don't cause the characters to be returned in reverse order).

A normal fetch will prove that this is what has happened. It reads forward from the divider.

```
{}-----  
{ } %FT,B/={HOUSE ON THE HILL}  
{ }  
{}-----
```

Thus an FDC with a negative value will Fetch Characters to the left of the divider, and a positive value will give characters to the right of the divider.

Another characteristic of the FDC primitive is the way the default argument reacts. The only time the value of an FDC command is the default argument is when the value of a fetch D characters is the null string, because no characters are left to read. If you did a

```
{}-----  
{ } %MD,B/=   
{ } %FDC,B, 50,Z/={THE HOUSE ON THE HILL}  
{ }  
{}-----
```

on "text" B, which was defined as THE HOUSE ON THE HILL, the result of the FDC command would be THE HOUSE ON THE HILL, not Z, even though it is obvious that there are not fifty characters in the "text" B.

Now that the "text" pointer is at the end of the "text", the result of another command would be the default argument.

```
{}-----  
{ } %FDC,B, 50,Z/={Z}  
{ }  
{}-----
```

The same characteristics apply to FDC commands with negative numbers of characters called.





LESSON EIGHTEEN - Fetching to matching substrings

There are a number of primitives that will Fetch parts of "texts" in accordance to various requirements. We shall show those that we find to be most useful. Additionally we will mention others we have heard of without describing them in detail in this lesson.

The first one of these is the FR (Fetch Right match) primitive; a description of its format and how it works follows:

```
| %FR,A,TEXT,Z/|
```

FR Fetch Right match

In this example, A is the name of some text in the text area, TEXT is the combination of characters to be searched for, and Z is the default argument.

The actual working of Fetch Right is a great deal like Fetch Element in that it fetches out elements of strings, but in FR the elements brought out are separated by the character combination signified in the third argument of FR. For example:

```
{ }-----
{ } %DT,Q,ADAM AND EVE/=
{ } %FR,Q,N/{ADAM A}
{ }
{ }-----
```

The value returned was the element of Q up to the letter N. The divider is moved to the character after N, or the D. Note that the default argument was left out of the FR primitive. If another FR with a default argument is executed, the following results:

```
{ }-----
{ } %FR,Q,N,Z/{Z}
{ }
{ }-----
```

The FR function gives all the text up to the next N. Since there is no further N, the default argument is given and the text divider is left in the same spot as before the FR command was executed.

However, if a regular Fetch Text is now done, D EVE is returned:

```

{}-----
{} FT,Q/={D EVE}
{}
{}-----

```

Remember, as in the other fetching primitives, if the elements brought out are executable expressions, be sure to use the ampersand "&" warning character.

In a similar manner the Fetch Left "FL" primitive may be used to fetch text from the current location of the text divider to a match to its left.

```
|%FL,A,TEXT,Z/|
```

FL
Fetch Left match

In addition the "FDM" Fetch D Match primitive can fetch right or left, depending on the sign past a desired number "D" of matches.

```
|%FDM,A,D,TEXT,Z/|
```

FDM
Fetch D Matches

This is one of the fancy types of Fetching primitives. The form of its use is shown below. What it means is that if it is used with reference to the selected "text", then the value is all it finds from the current location of the "text" divider to the first character in the "text" which is one of the characters in the string denoted by "X" in the below example.

```
|%FTB,NAME,X,Z/|
```

FTB
Fetch To Break character

As usual if nothing is returned the value will be Z, and if a match is found then the "text" divider is moved to a place after the last character returned from the "text" and points to the next character available to be read.

This second fancy Fetching primitive is very similar to the FTB primitive. The only difference is that the value depends on not finding rather than finding a character from among those denoted by "X" in the following example:

```
|%FTS,NAME,X,Z/|
```

FTS
Fetch To Span character

LESSON NINETEEN - Integer Arithmetic - "S" Syntax

The Add primitive AD has already been introduced, but not in its full format; here is a complete AD primitive:

AD - Add

```
|%AD,N1,N2,...,Nn/|
```

Just in case we forgot to mention it earlier, you can use positive or negative numbers: for negative numbers put a minus `| - |` sign (sometimes also called a hyphen) in front of the number, you do not need a plus `| + |` sign for positive numbers; we haven't tried it out but we don't think that a double negative should work to make the number positive although some people think it should as in the following example which was faked:

```
{ }-----
{ } %AD, 5, -----1/={4}
{ }
{ }-----
```

In fact the rule is that the scan for numbers starts at the right hand end of the number string going leftwards until it finds the first non numeric character. In the above example this is a `| - |` or hyphen or minus sign in this case; this makes the number after it negative and the symbols before it are considered to be alphabetic, that is to say an awful lot of hyphens.

There are three other primitives for Subtract, Multiply, and Divide.

The one for subtract has the mnemonic SU; in the first of the examples, 3 is subtracted from 14, with a value of 11 returned.

SU
Subtract

```
{ }-----
{ } %SU, 14, 3, 0/={11}
{ } %SU, 14, -3/={17}
{ } %SU, , 14/={-14}
{ } %SU, -14, -3/={-11}
{ }
{ }-----
```

Multiplication uses the mnemonic MU; in the first of the examples, zero is multiplied by 2 to receive an answer of zero.

MU Multiply

```
{}-----
{} %MU,0,2/={0}
{} %MU,12,-12/={-144}
{} %MU,-12,12/={-144}
{} %MU,-12,-12/={144}
{} %MU,0,-2/={0}
{}-----
```

Division has the mnemonic DI; in the example, 6 is divided by 2, the same as the mathematical symbol $6/2$ or the division of six by two. If you did not know, the answer is 3.

DI Divide

```
{}-----
{} %DI,6,2,Z/={3}
{}-----
```

When the DI primitive produces a non-integer value, as in `%DI,5,2/=2`, the result is rounded to the next smaller integer (in absolute value).

Perhaps, if you have used mathematical computer languages before, you will notice that all examples of arithmetic primitives contain integral numbers, commonly known as whole numbers (1). The SAM76 language uses integer arithmetic. As you go on, you will find that you can create floating point number systems or fractional number systems. The problem of non-whole numbers is not handled in any one standard way. Thus the "S" syntax of the SAM76 language offers the basic integer math and allows the user to design his own way of dealing with fractions. The user can even use procedures to produce his own number base.

The Z used in the Divide example is what is called a default argument. This is the result that is returned if the numbers are too large for the computer's memory to handle. This is called overflow. Some computers can handle numbers so big that a user would not have use for such a large value. Some computers do have smaller limits, though. This depends on the individual computing system. If the number resulting from an arithmetic command is too large, the default argument is returned to show that the computation failed. If the default argument consists of a procedure, the procedure will be executed (2).

For example, consider an extremely limited system which does not have an infinitely large memory and this expression is faked out:

```
{}-----
{} %DI, 57921, 0, !%OS, TOO LARGE///={TOO LARGE}
{}
{}-----
```

For purposes of illustration only we are assuming that the resulting value of the above expression would be too large for the machine, and the OS primitive would be executed and TOO LARGE would be printed out. Leaving out the default argument does not affect computation. It just implies that nothing will result if there is an overflow.

To handle more than two arithmetic functions in one SAM76 expression it is necessary to nest the primitives.

```
{}-----
{} %AD, 4, %AD, 3, 7///={14}
{}
{}-----
```

|4+(3+7)|

In the example, the innermost AD is executed first. That is why the mathematical representation has parens around the three and the seven; incidentally this is typical of the format used for the "A" syntax of the SAM76 language which will be described elsewhere.

Math problems like $((4+5) * 7)/(4 * 7)$ can be represented by nesting also.

```
{}-----
{} %DI, %MU, %AD, 4, 5/, 7/, %MU, 4, 7///={2}
{}
{}-----
```

Notice that this doesn't follow the math symbols from left to right. The SAM76 language expressions work inside to out, and handle the numbers two at a time in a manner very similar to the way people handle numbers.

An interesting characteristic of the arithmetic primitives is that of labels for numbers.

```
{}-----
{} %AD, DOG3, 4/={DOG7}
{}
{}-----
```

This means that an expression, like the example, can contain both alphabetic and numeric characters. Any characters that precede the numerals at the tail end of the second argument will be returned with the answer.

```

{}-----
{}  %AD, 3DOG1, 4/={3DOG5}
{}  %AD, DOG, CAT4/={DOG4}
{}  %AD, DOG-4, 5/={DOG1}
{}  %AD, APPLES 5, PEARS6/={APPLES11}
{}
{}-----

```

This property is useful for distinguishing the nature of different numeric strings by these labels.

Just in case you do not like to use decimal numbers the SAM76 language allows you to change the number base (if you have eight fingers you might like to use "base 8" although you might prefer "hex" but are not fortunate enough to have sixteen fingers). The format for changing the number base uses the mnemonic CNB, short as you can guess for "Change Number Base". This is illustrated thus:

CNB Change Number Base

```
|%CNB, 8/|
```

In the event you forget what number base you have selected you can use the QNB primitive which is the mnemonic for "Query Number Base" and returns as its value the current setting of the base for the integer arithmetic system:

QNB Query Number Base

```

{}-----
{}  %QNB/={10}
{}  %CNB, 16/=
{}  %QNB/={16}
{}  %CNB, 10/=
{}
{}-----

```

The number base or radix of the integer arithmetic system in the SAM76 language is referred to as the "N" base.

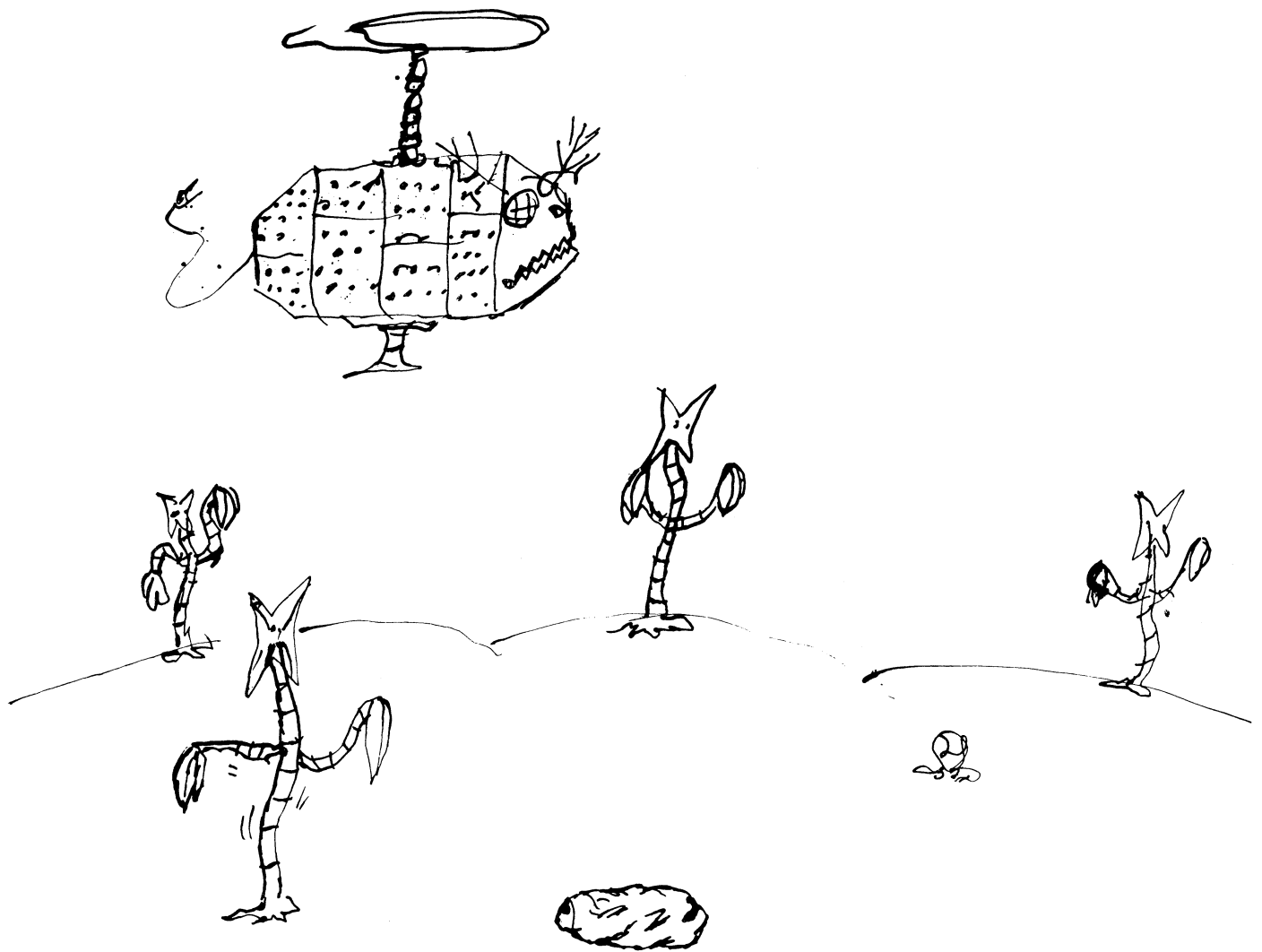
Note 1	According to the Student Mathematics Study Group there is no such thing as a negative integer. In this volume integers are considered as positive and negative.
--------	---

Note 2

This should really never happen with AD, SU, and perhaps under very special circumstances with MU. If it does you should execute the programmer who built the SAM76 processor you are using instead of the default argument. If you are wondering why this is being said just ask yourself why the computer should overflow if you don't when adding, subtracting or multiplying on paper. As for DI, the only reason for the default argument is that you would like to know if you are dividing by zero. With an old fashioned mechanical desk calculator this tended to wear the bearings out and the funny noises the machine then made was the equivalent of the default.

String Arithmetic





LESSON TWENTY - Choosing Sides Arithmetically

II is useful for testing string equalities character by character. Therefore one string of digits can be compared with another string of digits. But this is not all a mathematician may use to compare two numerical values.

Thus we have the primitive IG which is the mnemonic for "If Greater":

|%IG,A,B,T,F/|

IG
If Greater

In the example, a test is made to see if number A is algebraically greater than number B, and if it is, the result is T. If it isn't, the result is F. T and F may be strings, but it is most often that they are protected procedures. For example, suppose we wanted to see if multiplying three times three was greater than three plus three (of course you geniuses already know the answer). Then, if it was, to execute "text" named A, and if it wasn't, to execute "text" B. To do so, we would use this example:

|%IG,%MU,3,3/,%AD,3,3/,!%A//,!%B///|

Let us examine what happens in the example.

First the computer does the multiplication and the addition and the result would look like this expression:

|%IG,9,6,!%A//,!%B///|

Since everyone (?) knows that nine is larger than six, the value of the IG primitive is the contents of the fourth argument which is %A/. This example will thus execute %A/. If three times three had been smaller than or equal to three plus three, %B/ would have been executed.

```
{}-----
{} %AD, /= {0}
{} %MU, 5, /= {0}
{}
{}-----
```

Note should be taken of the fact that the null string argument becomes zero in all the arithmetic primitives, including "If Greater" than, thus

```
{}-----
{}  %IG, ,-1,T,F/={T}
{}
{}-----
```

is equivalent to

```
{}-----
{}  %IG, 0,-1,T,F/={T}
{}
{}-----
```

and as can be seen the result in both cases is T.

Although the SAM76 language does not usually provide other functions like "If Less Than" or "If Greater than or Equal to" it is easy to create user defined functions to achieve this using the IG primitive. This is illustrated below:

```
{}-----
{}  %DT,ILT,!%IG,q2,q1,(q3),(q4)///=
{}  %PT,ILT,q1,q2,q3,q4/=
{}  %VT,ILT/={%IG,[2],[1],([3]),([4])}/}
{}  %ILT,10,20,TRUE,FALSE/={TRUE}
{}
{}-----
```

Caution!	You are almost through with the Primer now. The next part is a bit tricky so if you feel the need, you had better stop. This will be the last warning but you can't really read straight through much farther anyway.
-----------------	---



LESSON TWENTY ONE - Viewing Texts

This lesson discusses a SAM76 primitive called VT short for "View Texts":

```
|%VT,T1,T2,...,Tn/|
```

VT
View Texts

This is the format of a View Text expression with T1, T2, ... , and Tn being the name of the "texts" in question.

What View Texts does is print out the strings named respectively T1, T2, ... , and Tn with all of their partitions marked and numbered. The value of VT is the null string.

```
{ }-----
{ } %DT,A,APPLE TREE/=
{ } %PT,A,E,T/=
{ } &FT,A/{APPL R}
{ } &FT,A,$,@&/{APPL$ &RSS}
{ }-----
```

This example defines the "text" named A as APPLE TREE, and then partitions out "text" A on E and T. A fetch to A now results in APPL R. But when a View Texts expression is executed on A,

```
{ }-----
{ } %VT,A/{APPL[1] [2]R[1][1]}
{ }-----
```

will be printed out.

Another capability of the VT primitive, is quite useful. This is the ability of VT to show the location of the "text" divider. Using the string A execute :

```
{ }-----
{ } %FE,A/{APPL}
{ }-----
```

The result will be APPL and the "text" divider will be left pointing to or preceding the next character to be read. In this case, it is a space. Executing a View Texts command at this point will cause

```

{}
{} %VT,A/={APPL[1][~] [2]R[1][1]}
{}

```

to be printed out. The up arrow between the open and close square brackets signifies the "text" divider. The "text" divider is shown as being in front of the space, instead of pointing to it, for the simple fact that it gets to be quite messy to try and type two characters in the same spot.

Thus the VT primitive can reveal the complete composition of a string for analysis or for any other purpose that comes to mind.



LESSON TWENTY TWO - Tracing

A technique that is used for analyzing scripts is called tracing, because it traces the steps taken in the evaluation of a SAM76 language expression. Since the result of the trace function is null, the `|%` form is used to activate the Trace Mode, and the `|&` form is used to neutralize or turn it off.

When the Trace Mode trace is in operation, the computer reacts in a non-standard manner. Each line feed causes the computer to evaluate the expression just displayed and to display the next expression to be evaluated. Since the computer reacts differently when it is tracing, there are two commands involved with the trace process.

TM
Trace Mode

```
|%TM,D/|
```

where "D" is an optional user specification as to the number of steps to be performed automatically when the trace is on, and

```
|&TM/|
```

in which the use of the `|&` form of the primitive is used to "neutralize" or turn the Trace Mode off.

Before an example of tracing is shown, it is important to show the special symbols printed out during trace. When scanning takes place with the trace function:

%	becomes { }
&	becomes {\}
,	becomes {^}
/	becomes { or \}

When tracing, it is important to remember the `%OS,%IS//` is loaded before anything else happens. As an example of a trace, the string A will be called which was defined by:

```
{}-----
{} %DT,A,!%DT,1,HELLO/%OS,%AD,3,4//&FT,1//=
{}-----
```

Every time the line-feed key is struck, Trace goes through executing another step. In the example which follows, the left hand column shows what is printed at the user's teleprinter. To the right is a representation of the working area.

Starting at the top, the restart expression %OS,%IS// is scanned, and it is the IS which is activated. This is represented by the IS printed out. When the user types a line-feed code, the command IS is executed (the system reads until the Activator is hit) and the next step is printed out. In this case, it is the %FT,A/ that was typed in for the IS.

The trace mode will continue until an &TM/ is executed or any character other than line-feed is hit (unless, of course, the other characters were typed during the execution of an IS or IC primitive).

Trace is extremely valuable when one must examine the workings of a complex, or even not so complex, script. A good exercise in script-analysis is to use the scripts from the applications section of this book and trace them. Try to look carefully at these scripts and figure out how they work with the help of trace.

The normal manner of using the Trace Mode was described earlier. There is another type of Trace Mode which gives a broader picture of what is happening in the computer. This broader picture can be seen by executing the TMA, Trace Mode All function thus:

TMA
Trace Mode All

```
{}-----
{}  %TMA/=
{}
{}-----
```

This type of presentation will stay in effect until the Trace Mode All option is turned off or neutralized by doing the following:

```
{}-----
{}  &TMA/=
{}
{}-----
```

This is possible because the value in both cases is the null string, and the function itself only turns a switch on or off in the computer.

A detailed description of this broader mode is not given because it is more complicated to explain and uses up too much space in this book; just try it out and you will get the picture right away.

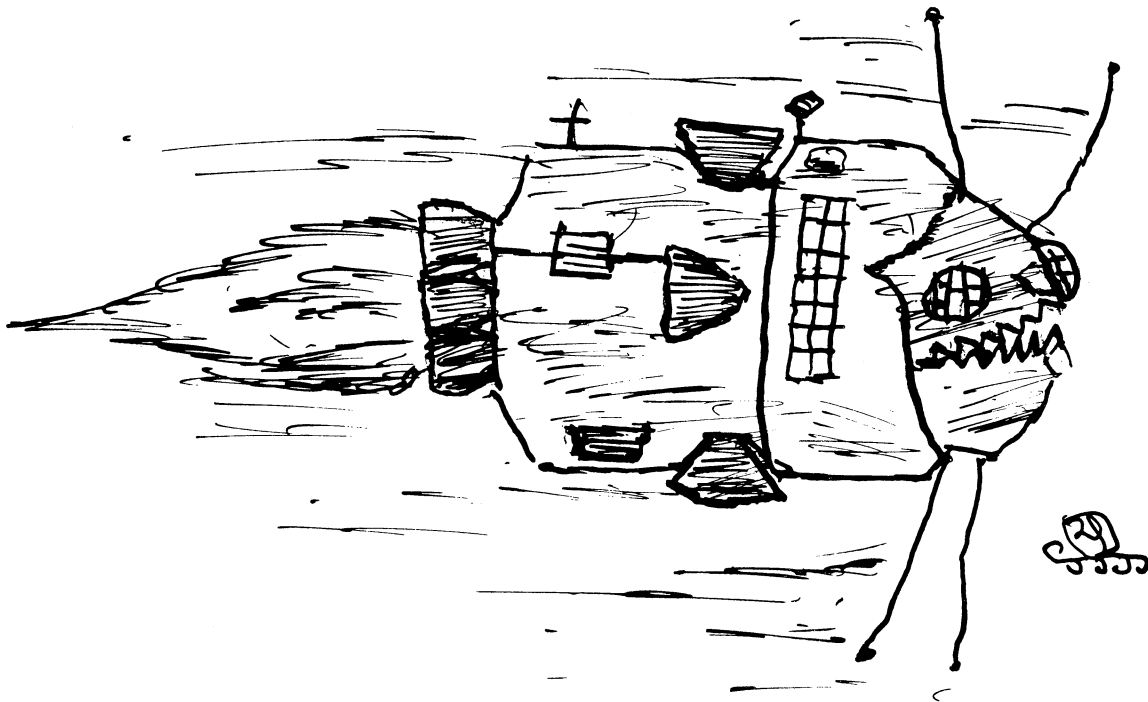
Example of use of Trace Mode

```

{}-----
{}
{}  %TM/=                               Typed in by the user
{}  |OS^|                               Restart Expression prints
{}                                     null string, new copy of
{}                                     |%OS,%IS/| is loaded
{}
{}
{}  |IS|                               |%IS/| of Restart Expression
{}                                     is now going to wait
{}                                     for user to do input
{}  %FT,A/=                             Typed in by the user
{}  |FT^A|                             |%OS,%FT,A//|
{}
{}  |DT^1^HELLO|                       |%OS,%DT,1,HELLO/|
{}                                     |%OS,%AD,3,4//|
{}                                     |&FT,1//|
{}
{}  |AD^3^4|                           |%OS,%OS,%AD,3,4//|
{}                                     |&FT,1//|
{}
{}  |OS^7|                             |%OS,%OS,7/|
{}                                     |&FT,1//|
{}  {7}                                Character {7} output by OS
{}  \FT^1\                             |%OS,&FT,1//|
{}
{}  |OS^HELLO|                         |%OS,HELLO/|
{}  {HELLO}                            Text output by OS command
{}
{}  |IS|                               |%OS,%IS//| New copy reloaded
{}  &TM/=                             Typed in by the user
{}  \TM\                             |%OS,&TM//| Turns trace off
{}-----

```

The system now returns to normal nontracing mode.



LESSON TWENTY THREE - Logical Bit Manipulation
--

The following primitives may be extremely confusing to the reader who has never had any Logic. For those who have not, and would not want to be bothered with any, it may be best to skip this section until the need arises. Therefore don't throw this book away, for who knows when you may run into problems involving logical elements!

For those who wish to learn how logical operations are handled in the SAM76 language, the following was written.

There are five Logical algebra primitives. We shall illustrate how all of these work using the octal digit system, which represents a three binary digit byte by its octal digit, or base 8 digit.

Base-8	0	1	2	3	4	5	6	7
Base-2	000	001	010	011	100	101	110	111

The first of the logical functions is NOT, logical NOT also sometimes talked about as the "complement".

|%NOT,x/|

NOT
(complementing)

Here the symbol "x" stands for an octal digit (octal number) that has any number of octal digits. This number is complemented octal digit by octal digit according to the rule:

Digit	0	1	2	3	4	5	6	7
Complement	7	6	5	4	3	2	1	0

For example:

```

{}-----
{}  %NOT,457601/={320176}
{}
{}-----

```

The second logical primitive is AND, or logical AND.

```
|%AND,x1,x2/|
```

AND (uniting)

x1 and x2 are general octal digits whose binary representations are used to form a union, otherwise known as "inclusive OR".

As an example, this primitive and its result are shown next to a binary representation:

```
|000 100 101 111      %AND,457,3214/={3657}
|011 010 001 100
|011 110 101 111
```

Note that the shorter number had leading zeros added. This causes each string to be the same length.

The third such primitive is OR, or logical OR.

```
|%OR,x1,x2/|
```

OR (intersecting)

x1 and x2 in this case are used to form the binary intersection of the two values, otherwise known as the AND. The value returned is an octal digit. An example is shown in the same format as AND:

```
|101 011 001      %OR,531,24/={20}
|000 010 100
|000 010 000
```

The result string is the same length as the shorter argument string.

The fourth logical primitive is ROT or logical ROTate.

```
|%ROT,D,x/|
```

ROT (rotating)

D is the number of bits rotated, plus for rotate left, minus for rotate right, and "x" is the value to be rotated. Bits that are moved off the line at one end reappear on the other end of the line. The value returned is the rotated number. For example:

```
|100 001 101 011      %ROT,2,4153/={0656}
|000 110 101 110

|100 001 101 011      %ROT,-2,4153/={7032}
|111 000 011 010
```

Thus the same number of octal digits is returned every time.

The fifth and last function is Logical Shift, SH short for Shift.

|%SH,D,x/|

SH (shifting)

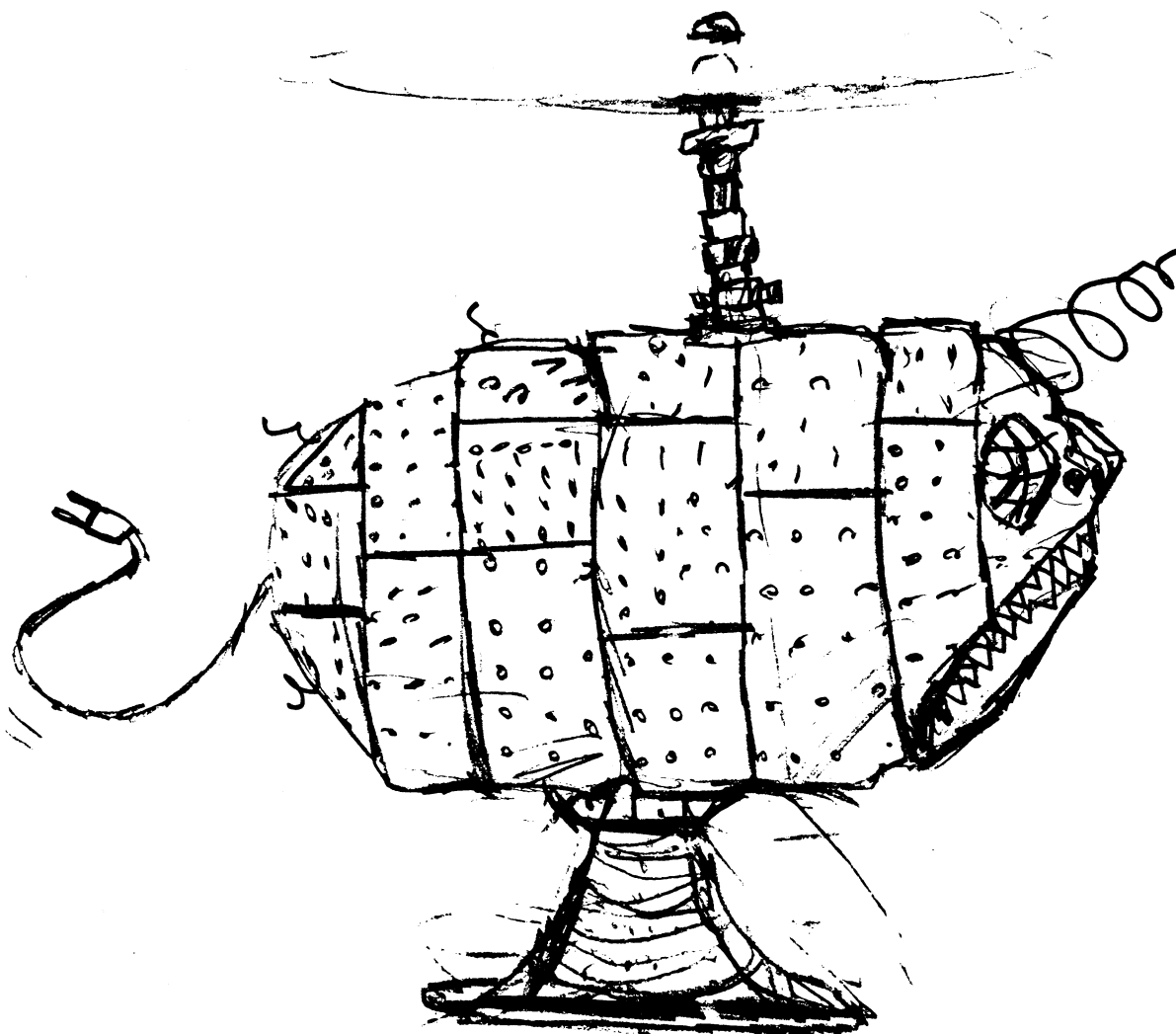
This shifts right or left depending upon the sign of the second argument. Logical Shift operates in the same manner as logical rotate except that bits shifted past the end of a line are lost. For example:

100 001 101 011	%SH,2,4153/={0654}
000 110 101 100	

100 001 101 011	%SH,-2,4153/={1032}
001 000 011 010	

As in ROT the number of octal digits returned is the same as the number of octal digits used originally.

In the logical primitives, as in arithmetic, the numbers are scanned from the right to the left, stopping on the first non-numeric character. Any non-numeric placed before these values are deleted.





LESSON TWENTY FOUR - Filing Texts in Auxiliary Storage
--

Computers do not always have enough memory to hold all the scripts needed to do some jobs, therefore the memory of a computer is often supplemented by another type of memory called bulk storage. This may be magnetic tape, magnetic disk, magnetic drum, or some other device. These are all similar in the respect that they are fairly slow and deal with large pieces of memory; often as much as the total memory of the computer. This mechanism is obviously useful since scripts can be stored for later use. Jobs that would be too large for the machine if it didn't have supplementary memory can be done on the machine.

Historical Note	Without wishing to give the appearance of bragging we think that you should know that we pioneered back in 1968 in the field of hobby computing and actually demonstrated and used an inexpensive (\$35) cassette deck bearing the "Raytheon" name in our exhibit at the Spring Computer Conference. Several companies thought this was for real and asked Raytheon about it receiving various weird and strange answers having to do with their educational division. At the next Computer conference there were several cassette data deck offerings and now ...
--------------------	--

Since storing information in bulk storage is important, there must be a primitive to do this. The operation of storing information in bulk storage is done by the primitive Store File, otherwise known as SF.

SF Store File

|%SF,A,T1,T2,...,Tn/|

The following example causes the creation of a file named DOG 1, which consists of the defined strings 1, 2, 3, and A to be stored in the bulk storage device:

```

{}-----
{} %DT,A,WHAT/=
{} %DT,1,APPLE/=
{} %DT,2,ORAHPE/=
{} %DT,3,PEAR/=
{} %SF,DOG 1,1,2,3,A/=
{}
{}-----

```

Note that Store File can store in one operation. Store file affects memory in the following manner.

When an LT is executed after the SF is completed, the strings referred to in the SF will be found to be missing.

In addition to the method of using the Store File primitive described above, you can also use it to store all of the "texts" in the computer without having to know what their names are. This is done by just not giving any names of "texts" at all thus:

```
{}-----
{} %SF,DOGSALL/=
{}
{}-----
```

Note that there is no comma after the file name "DOGSALL". If there were such a comma as in the example that follows, the SF primitive will store in the file whose name will be DOGSALL the one and only possible string in the computer whose name is the null string:

```
{}-----
{} %SF,DOGSALL,/=
{}
{}-----
```

The primitive that does the opposite of Store File, in that it brings back the strings into standard memory from bulk storage, is called Bring Files, or BF.

BF
Bring Files

```
{}-----
{} %BF,DOG 1/=
{}
{}-----
```

In the example, DOG 1 is the name of the file to be brought in from bulk storage. It copies the strings in the file back into standard memory; that is, after a Bring Files is executed, the file in storage still exists.

An interesting feature of the Bring Files command is that it will bring the file (if it actually exists) and place the "texts" in that file into the computer regardless of their names, or of pre existence of like names. This is another way in which you can have more than one "text" in the "text area" with the same name; this may sound as if it could cause all kinds of trouble; it can - but this capability is more useful than troublesome and it is up to you to know what you are doing.

The next one of these functions is Erase Files, known as EF. This acts like ET in that where ET erases "texts" from the computer's "text" area, EF erases files from bulk memory.

EF
Erase Files

```
|%EF,F1,F2,...,Fn/|
```

The example shows the format of EF, where F1,F2,...,Fn are the names of the files to be erased. When an erase file is executed, the file is removed from bulk memory.

In order to find out the names of the files that may exist in auxiliary storage the SAM76 language provides the LF primitive. LF is the mnemonic for List Files and works exactly like the LT "List Texts" function except that the list is that of Files and not of Texts. The normal format for the LF primitive is:

LF
List Files

```
|%LF,XXX/|
```

In the above XXX represents the string of characters you wish to have returned preceding each file name.

In this manner by being clever you can create a function which is very dangerous because it will erase all files. We will call this user defined function "EAF" and it is built as follows:

```
{ }-----
{ } %DT,EAF,!%EF%LF,<,>////=
{ }
{ }-----
```

If you should be bold enough to define such a procedure never do the following unless you really mean to wipe yourself out:

```
{ }-----
{ } %EAF/=
{ }
{ }-----
```

A special combination function called UF short for Update File is provided in the SAM76 language. This function combines the actions of EF "Erase Files" and SF "Store File". The reason that this is useful is that it actually first stores the file, then if that is successful it will erase the file that originally had the same name. This is so that if someone turns the lights out in your room just after you have done the Erase Files, and before the Store File, all the work you might have accumulated in your computer memory will not suddenly vanish because you plugged your computer in the same circuit as the lights, and you do not have emergency battery back up. Just for reference the format is:

UF
Update File

```
|%UF,FILENAME,T1,T2,...,Tn/|
```

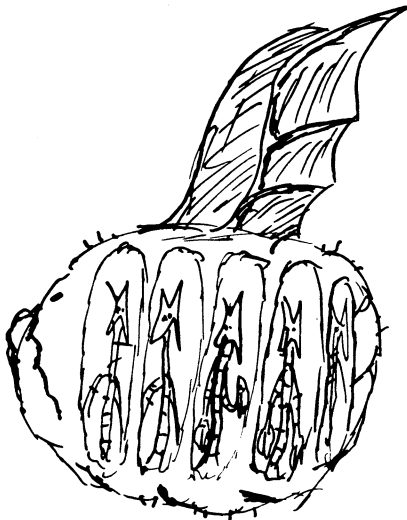
The same feature of being able to store all of the "texts" without naming them that was described in connection with the SF primitive applies for UF also.

The File Branch function, whose mnemonic is FB allows you to check the auxiliary memory for the presence of a file without actually having to bring it in memory, or to analyze the results of doing a List File function. The format is similar to that used for the TB, II and IG functions in that if the condition is true you get one value back, if false you get the other. You specify in the expression the values you wish to have if true or false thus:

FB
File Branch

```
|%FB,FILENAME,VALUE IF TRUE,NOT PRESENT/|
```

In the above if the file whose name is FILENAME is in auxiliary memory then the expression will be replaced by the character string "VALUE IF TRUE", if not there the "NOT PRESENT" is the value of the expression. Needless to say but we will to make sure you remember, these two options can be expressions or procedures.



LESSON TWENTY FIVE - Changing the Activator

Perhaps you have wondered why the equal |=| sign was so arbitrarily chosen to be the Activating Character. Perhaps you have not. In any case this character can be changed. Appropriately enough the command CA for Change Activator was chosen to do this. It is used in the format:

CA
Change Activator

|%CA, s/|

where the first character of the string "s" becomes the new Activator. The equal sign may or may not be the activator in this case.

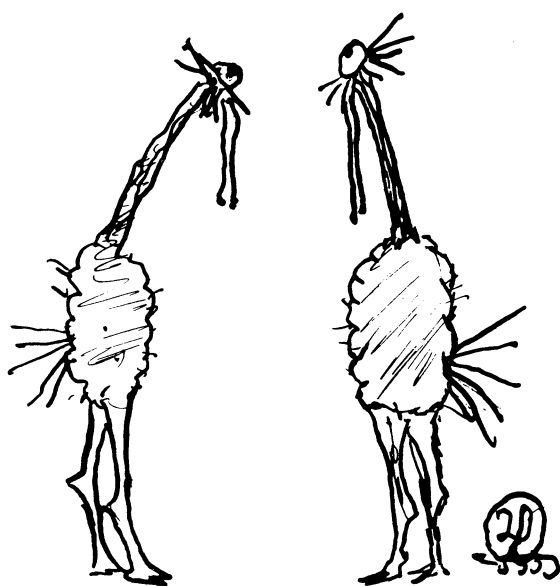
When the Activator is changed, it remains that way until it is changed back. For example:

```
{}-----
{} %DT,A,APPLE/=
{} %FT,A/={APPLE}
{} %CA,?/=
{} %FT,A/?{APPLE}
{} %CA,=?
{} %FT,A/={APPLE}
{}
{}-----
```

If one wanted to be totally evil, one could execute this example:

```
{}-----
{} %CA,&IC//=,
{}
{}-----
```

Now the SAM76 language would be very unusable because with the Activator being a comma, it would be most difficult to do anything which has arguments like the Change Activator command. There are other characters which can do likewise, but these are left to the reader to figure out.



LESSON TWENTY SIX - Functions using the "X" number base

Since you may wish to use other than octal notation, the SAM76 language provides for changing the base for the logical functions as well as any other functions which deal with "binary" or bit strings. The number base for these is identified as the "X" base (in the arithmetic functions the base was identified as the "N" or number base). The function used to change the "X" Base has as its mnemonic CXB the format of which is usually:

CXB Change "X" Base

```
|%CXB,d/|
```

where "d" is a number in base ten.

The QXB, Query "X" Base primitive allows you to find out at any time what is the current setting of the "X" base; (this corresponds to the QNB for the arithmetic functions). The values returned by the examples shown below are the current setting in base ten:

QXB Query "X" Base

```
{ }-----
{ } %QXB/={8}
{ } %CXB,16/=
{ } %QXB/={16}
{ } %CXB,8/=
{ }
{ }-----
```

The purpose of this function is to allow you to actually see what is stored in any memory location in the machine. The format is:

XR eXamine Register

```
|%XR,xadr/|
```

In the above "xadr" is a number in the current "X" base which denotes the address in computer memory. The value of this function is the contents of that location..

This function is primarily oriented to machines which group data in two consecutive memory locations, and where the user wishes to have the contents displayed as a single number in a range equal to the sum total of bits. The format is shown thus:

XRP
eXamine
Register Pair

|%XRP,adr/|

In the above expression "xadr" denotes the lower of two consecutive memory addresses the user wishes to examine.

This function is used in the following format to allow you to enter any desired number expressed in the current "X" base into any memory location:

XW
eXperimental
Write
in register

|%XW,xadr,x/|

in the above, "x" denotes the value to be entered at the location whose adress is denoted by "xadr".

This function allows entering data which is denoted by "x" in the sample format into the consecutive pair of memory locations denoted by "xadr". If the % form is used then the data is entered right justified in the lower of the two addresses; if the & form is used the data is entered so that the low order byte is placed in the higher of the two memory addresses:

XWP
eXperimental
Write
register Pair

|%XWP,xadr,x/|

This very clever function allows you to make an unconditional jump at the time it is executed to the adress denoted in the following format by "xadr":

XJ
eXperimental
Jump

|%XJ,xadr/|

This jump is so made that the right kind of information is saved to allow you to reenter at the same point if you should so desire.



LESSON TWENTY SEVEN - Counting up - down and around

Very often the need exists to do something some predetermined number of times. A counter is required and can be obtained in several ways.

The first way that will be illustrated makes use of a procedure defined by the user employing the AD and IG functions thus:

```
{}-----
{} %DT,COUNTER,!%DT,N,%AD,-1,%N///%IG,%N/,,(%OS,
{} %N///%COUNTER/),(%OS,
{} COUNT IS DONE/)///=
{}
{}-----
```

In the procedure named "COUNTER" defined above the first thing that happens is that the value of "text" N is reduced by a count of one; then using the IG primitive a test is made to see if the new value of N is greater than the "null" string (which is equal to 0), if it is greater then the current value of N is printed out and the whole thing repeated. If it is not (and 0 is not greater than 0) then the message COUNT IS DONE is the output.

To start things out an initial value for N is required; five is picked in the example below which shows the whole sequence:

```
{}-----
{} %DT,N,5/%COUNTER/=
{} {4
{} 3
{} 2
{} 1
{} COUNT IS DONE}
{}
{}-----
```

Other ways using procedures can be devised, some of them using a technique called "recursion". The technique illustrated above is called "iteration" (*).

.[23 |the "Z" registers

Because counting is so important and used so often the SAM76 language provides a family of functions that help with counting much faster than procedures and without cluttering up the "text" area with temporary "texts".

The first of these functions allows the user to preset a count in one of the various "Z" registers. This is done using the format shown below:

```
|%ZS,r,n/|
```

ZS
Z register Set

In the above, "r" is the register identification, and "n" the number to be initially loaded. This is the equivalent of the %DT,N,5/ in the first example of this lesson.

The second function allows the querying the register as to what its current value might be; this is done according to the format:

```
|%ZQ,r/|
```

ZQ
Z register Query

This is a combination function which increments the contents of the specified register, and depending on whether the new value therein is less than 0, equal to 0, or greater than 0, one of three values is returned:

```
|%ZI,r,less than,equal,greater than/|
```

ZI
Z Increment and branch

In the above format, "r" denotes the desired register and the three statements identify the arguments which are returned if the count is less than zero, equal to zero or greater than zero.

This function is very similar to the ZI function, except that the contents of the specified register is decremented by a count of one before the test comparison to 0 is made:

```
|%ZD,r,less than,equal,greater than/|
```

ZD
Z Decrement and branch

An example using the "Z" functions to do the same thing as the first example of this lesson is given below.

```
{}-----
{} %DT,COUNTER,!%ZD,1,,(%OS,
{} COUNT IS DONE/),(%OS,
{} %ZQ,1//%COUNTER/)///=
{}
{}-----
```

The procedure "COUNTER" has been defined using ZD and ZQ; now to get things started we do the following:

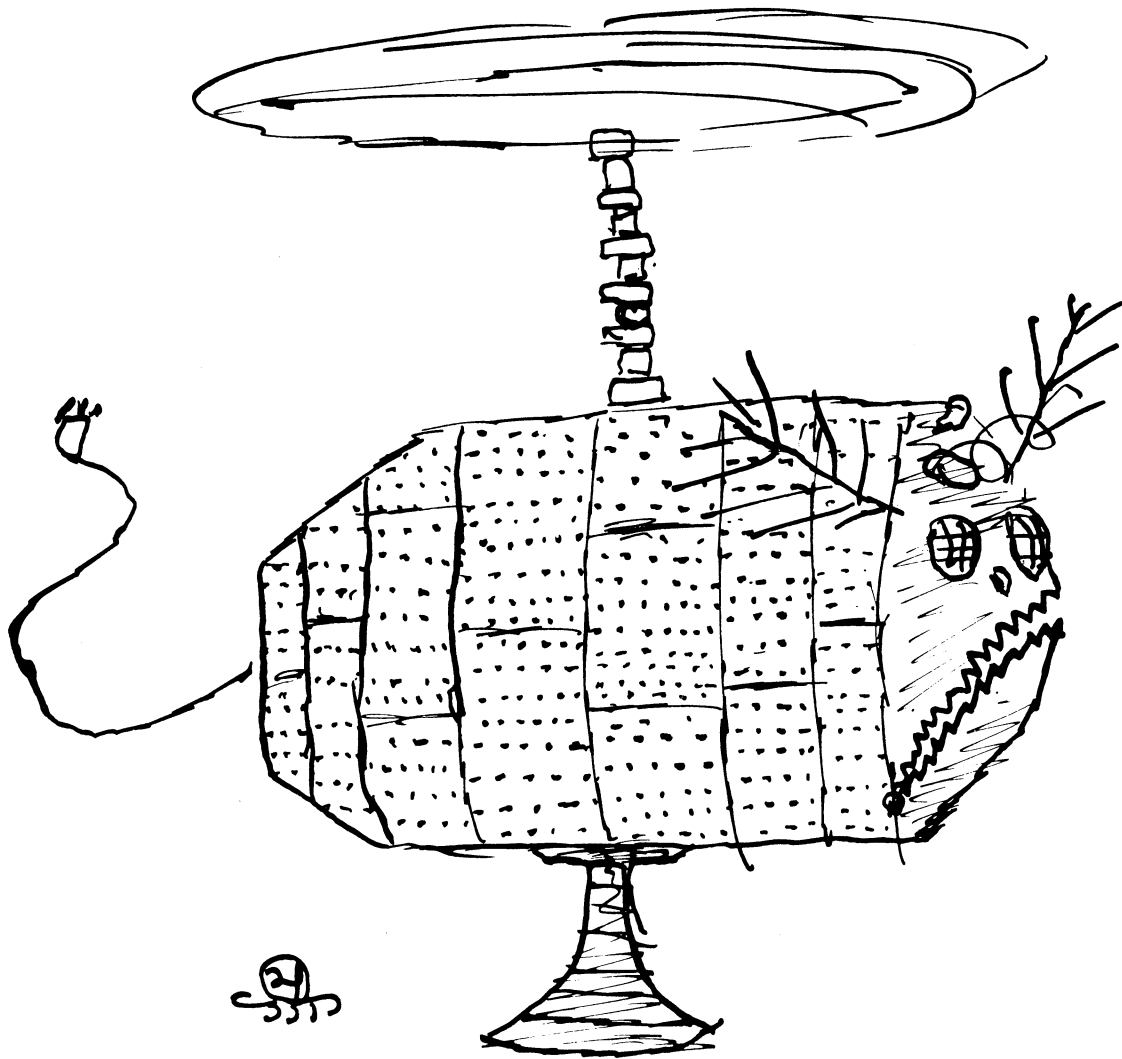
```
{}-----  
{ } %ZS,1,5/%COUNTER/=  
{ } {4  
{ } 3  
{ } 2  
{ } 1  
{ } COUNT IS DONE}  
{ }  
{}-----
```

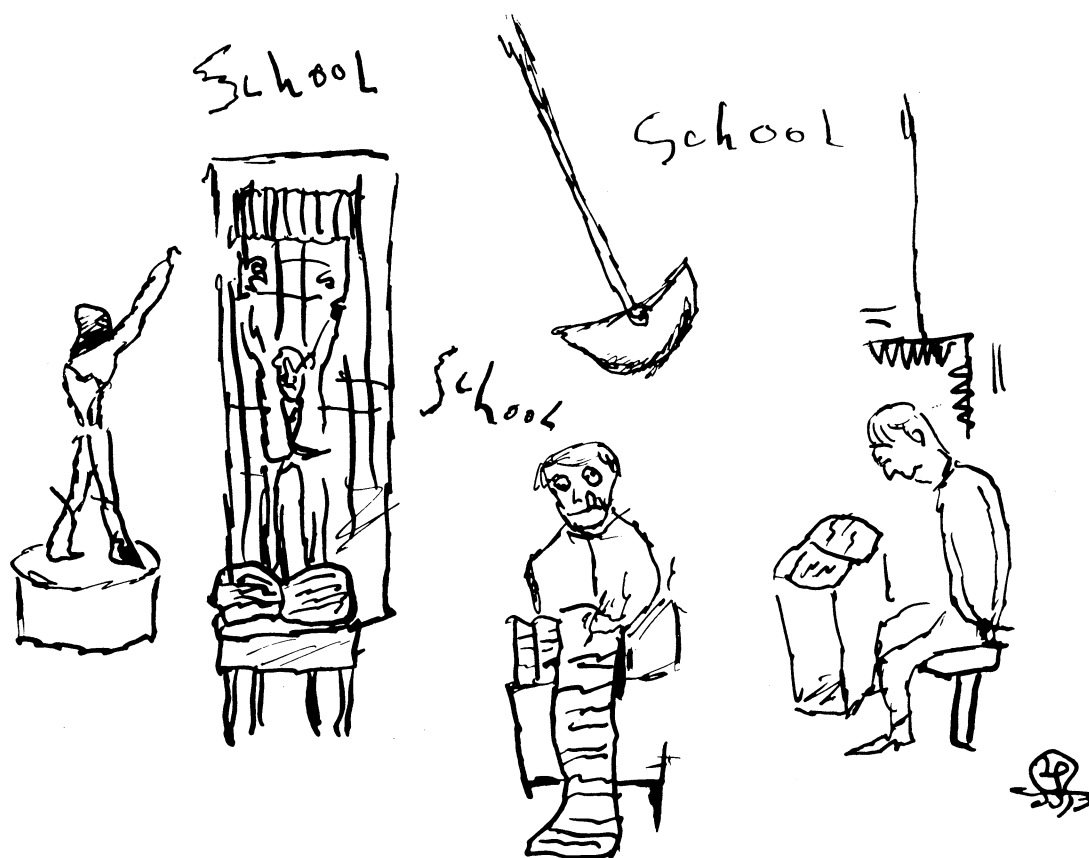
In addition to the fact that a great deal of time is saved in the computer by not having to constantly fetch and redefine the "text" N of the first example, another obvious benefit is that there is much less typing to do with the second example than had to be done with the first.

In the first example 85 keystrokes on the keyboard were required to define the procedure "COUNTER", whereas only 69 are required using the ZD primitive. This means that whether you type well or poorly the second method should only take two thirds of the time to write than the first method.

Perhaps of less importance to some people it should be noted that the second example takes less room in the machine "text" area.

Note	"To Iterate is human; to Recurse Divine".
*	{L. Peter Deutsch}





Foreword

Learning should be fun. All too often the learning process is severely curtailed, if not totally destroyed, by the imposition on potential learners of a poorly designed or a poorly implemented teaching process.

Learning and being taught are far from being synonymous. To the learner:

LEARNING IS FREEDOM BEING TAUGHT IS SLAVERY.

The former is reinforced by motivation and the pleasure of accomplishment. The latter fosters negative attitudes and if the deserved rebellion is successfully repressed by the teaching authority the only result is apathy, prejudice, and bigotry.

When the teaching authority thinks the subject matter being offered a potential learner is complex, this outlook is communicated to the student. Since the learner does not know enough about the material to form an opinion as to his or her ability to achieve new skills or insights the first thing well learned is that the rest of the material will be difficult to tackle. Unless strongly motivated, the student cannot but fail to derive any benefits from his teacher.

Equally disastrous to the learning process is the failure of the teacher to properly communicate to the student the value, meaning, and potential payoffs, either intellectual or economic, of successful completion of a course of study. Such a failure can, and frequently will, result in failure even when the subject matter is of the simplest form and the student already knowledgeable or even expert. In fact students may unlearn or fail to make use of previously learned skills and information if they are convinced that the whole process they are being subjected to (being taught) is irrelevant or lacks purpose or values.

This book is not designed to or intended to teach. It is designed to encourage learning in an area which all too often is considered by the teaching profession at the secondary school level to be beyond the capabilities of the students. This point of view is further coupled with the disastrous lack of understanding of the uses, purposes, and values of a technology which has come into being, grown, and reached today's level of penetration in our society since the time most of them started to claim the title of teacher.

I have heard it said that teaching would be fun if it were not for the students. This may be to the point, even more so since students are virtually unanimous in their opinion that learning is fun (and hence successful) without these same "teachers."

This is fully demonstrated by the existence of this book. Substantial portions were derived, with permission, from material written by several members of a student group which called itself the RESISTORS. I was associated with this group through its use of my barn as a play-house. This play-house is well stocked with a broad range of technical toys including such esoteric ones as three fully operational commercially produced electronic digital computers.

To the members of the group computers were not mystical, mysterious entities to be feared or ends in themselves. They were useful and pleasurable tools to be used when and where appropriate in their play-projects.

The students, who ranged from ten years of age through that at which most complete their high school education, could not in any manner be considered as the intellectual elite. The most that could be honestly said about them is that they generally fell in the upper third or quarter of their class in a school district not distinguished by any notably high rate of admission of its graduates to those institutions of higher learning which have particularly strict academic entrance requirements.

The authors of this book embarked on this project for two main reasons. One reason is that this venture was expected to provide the equivalent of a summer job, with the added attraction of freedom of schedule allowing for other equally, or more important activities such as scuba diving and glass blowing. A second reason was perhaps the feeling that they could - because of their immaturity, lack of experience, and limited knowledge - succeed in producing a text which through its primitive clarity born of its authors' youth, would be effective in communicating to the reader the method of use and the utility of an esoteric computer language.

There was a third reason, which although not official did act to spur their efforts, namely the desire to "psych out" the denizens and authorities of their respective school systems.

It is with great pleasure that I am able to dedicate this book to Barry, Dave, Gnat, John, and Joe. I hope that wherever they are today, they will recognize their own work and approve of the transformation that was made to finally bring the material to publication.

I hope that this book may serve to inspire some of you to allow yourselves to be trapped by a group of motivated, active students and embark on projects which are stated to be unquestionably beyond the capabilities of the students' "little" minds.

Claude A.R. Kagan

Hopewell Township, N.J. - Summer 1970 to Winter 1978

[illegible]

SAM76 Language Handbook - Contents Part 1

ii	DISCLAIMER and Copyright
iv	Acknowledgements
iv	R.E.S.I.S.T.O.R.S. Acknowledgements
vi	FOREWORD
vii	LESSON ZERO
1-1	Part 1 - Rudimentary
1-3	LESSON ONE - Syntax
1-5	OS Output String
1-6	Note *
1-7	LESSON TWO - Scanning
1-9	IS Input String
1-13	LESSON THREE - The Restart Expression
1-15	DIAGRAM
1-15	Cool It!
1-17	LESSON FOUR - Defining Texts
1-17	DT Define Text
1-19	LESSON FIVE - Fetching Texts
1-19	FT Fetch Text
1-21	LESSON SIX - Protection
1-21	Protecting Character Pairs
1-25	LESSON SEVEN - Procedures
1-29	LESSON EIGHT - FT Revisited
1-32	Note *
1-32	Take Ten!
1-33	LESSON NINE - Listing Text names
1-33	LT List Texts
1-34	Note *
1-37	LESSON TEN - Erasing Texts
1-37	ET Erase Text
1-38	EA Erase All
1-39	LESSON ELEVEN - Testing for Identities
1-39	II If Identical

1-45	LESSON TWELVE - Interaction
1-51	LESSON THIRTEEN - More Interaction
1-51	IC Input Character
1-51	ID Input D Characters
1-54	Note *
1-54	Hold it!
1-55	LESSON FOURTEEN - Partitioning Texts
1-55	PT Partition Text
1-59	VT Viewing Text
1-60	PT ampersand form
1-61	Stop!
1-63	LESSON FIFTEEN - Partitioning Texts - cont'd
1-67	Pause for ... !
1-69	LESSON SIXTEEN - Fetching Elements
1-69	FE Fetch Element
1-70	MD Move Divider
1-71	FDE Fetch D Elements
1-73	LESSON SEVENTEEN - Fetching Characters
1-73	FC Fetch Character
1-73	FDC Fetch D Characters
1-77	LESSON EIGHTEEN - Fetching to matching substrings
1-77	FR Fetch Right match
1-78	FL Fetch Left match
1-78	FDM Fetch D Matches
1-78	FTB Fetch To Break character
1-78	FTS Fetch To Span character
1-79	LESSON NINETEEN - Integer Arithmetic - "S" Syntax
1-79	AD - Add
1-79	SU Subtract
1-79	MU Multiply
1-80	DI Divide
1-82	CNB Change Number Base
1-82	QNB Query Number Base
1-82	Note 1
1-82	Note 2
1-85	LESSON TWENTY - Choosing Sides Arithmetically
1-85	IG If Greater
1-86	Caution!
1-87	LESSON TWENTY ONE - Viewing Texts
1-87	VT View Texts
1-89	LESSON TWENTY TWO - Tracing
1-89	TM Trace Mode
1-90	TMA Trace Mode All

1-91	Example of use of Trace Mode
1-93	LESSON TWENTY THREE - Logical Bit Manipulation
1-93	NOT (complementing)
1-93	AND (uniting)
1-94	OR (intersecting)
1-94	ROT (rotating)
1-94	SH (shifting)
1-97	LESSON TWENTY FOUR - Filing Texts in Auxiliary Storage
1-97	Historical Note
1-97	SF Store File
1-98	BF Bring Files
1-98	EF Erase Files
1-99	LF List Files
1-99	UF Update File
1-100	FB File Branch
1-101	LESSON TWENTY FIVE - Changing the Activator
1-101	CA Change Activator
1-103	LESSON TWENTY SIX - Functions using the "X" number base
1-103	CXB Change "X" Base
1-103	QXB Query "X" Base
1-103	XR eXamine Register
1-103	XRP eXamine Register Pair
1-104	XW eXperimental Write in register
1-104	XWP eXperimental Write register Pair
1-104	XJ eXperimental Jump
1-105	LESSON TWENTY SEVEN - Counting up - down and around
1-105	ZS Z register Set
1-106	ZQ Z register Query
1-106	ZI Z Increment and branch
1-106	ZD Z Decrement and branch
1-107	Note *
a-1	Appendages
a-3	Foreword


```

194 - qin,s0 t1,t2,...,t1
195 - cfc,d1,s\
196 - qfc,s0\
197 - qrd,t1\
198 - qrd,t1\
199 - sem,dev\
200 - \cxb,d\
201 - \cxb,d\
202 - qof\
203 - cro,s1\
204 - \cro,s1\
205 - qta\
206 - ea,t1,t2,t1\
207 - ed,t,d1,d2,vz\
208 - dq,s1\
209 - nu,s1,s2,...,s1\
210 - fts,t,s,vz\
211 - fts,t,s,vz\
212 - hc,s1\
213 - lw,n\
214 - lw,s0,s1,s2,...,sn\
215 - ra,d,s1,s2,s3,...,s1\
216 - lf,s0,d1\
217 - qfs,filename\
218 - uf,t1,t2,...,t1\
219 - qfe\
220 - bf,f,vz\
221 - sfe,extension\
222 - sf,t1,t2,...,t1\
223 - sdu,dir\
224 - ef,f1,f2,...,f1\
225 - qdu\
226 - fb,f,vt,vf\
227 - qcs\
228 - lff,s0\
229 - xrs,unit,track,sector,s0\
230 - xrs,unit,track,sector,s0\
231 - sw,s1,s2,s3,...,s1\
232 - sy,s1,s2,...,s1\
233 - dif,filename\
234 - dof,filename\
235 - rfr\
236 - wfr,s1\
237 - wt\
238 - ef,s0\
239 - en\
240 - en,current,new\
241 - lic,s0\
242 - loc,s0\
243 - rf,filename\

```

```

Query Id Number
Change Fill Character schema
Change Fill Char. (initial)
Query Fill Character schema
Query Left of Divider
Query Right of Divider
Set "Echoplex" Mode active
"Echoplex" Mode inactive
Change "X" Base (active)
Change "X" Base (initial)
Query "X" Base
Query Over Flow conditions
Change Rub Out char. schema
Change Rub Out (initial)
Query Rub Out char. schema
Query Text Area used
Erase All excepting
Extract "D" characters
Define Quote
Null
Fetch To Break character
Fetch To Span character
How Many Characters
Input Wait
List Where
Return Argument
List Files
Query File Size
Update File
Query File Extension
Bring File
Set File Extension
Store File
Select Directory Unit
Erase Files
Query Directory Unit
File Branch
Query Command String
List File Functions
X Read Sector
X Write Sector
Switches
System functions
Designate Input File
Designate Output File
Read File Record
Write File Record
Who is processor Title
Who are processors
Who is processor ser. Number
Change function Name
List Input Channels
List Output Channels
Read File

```

```

244 - wf,filename,s1
245 - sic,sym\
246 - soc,sym\
247 - rj,s,s1,d,s2\
248 - rp,s,s1,d,s2\
249 - etb,s1\
250 - cwc,s1\
251 - \cwc,...,\
252 - qwc,q2,a1,...,a1\
253 - rjn,n\
254 - xqs,s0\
255 - xi,port\
256 - xo,x,port\
257 - ti,s1,s2\
258 - stl,t1,t2,t3\
259 - da,s0\
260 - sda,da,mo,yr\
261 - cws,d1\
262 - \cws,x\
263 - \qws\
264 - rep,d1,d2,s1\
265 - qio\
266 - sio,lobyte\
267 - cpc,x1,t1,...,tn\
268 - qpc,s0,t1,t2,...,t1\
269 - nud,func,arguments\
270 - xrs,unit,trk,sec,s0\
271 - xws,unit,trk,sec,x1\
272 - xu,sub,arguments\
273 - xcf,s,x1\
274 - trs,sub,arguments\
Write File
Select Input Channel
Select Output Channel
Return justified lines
Return padded lines
Erase trailing blanks
Change Warning Character
Change Warn. Char. (initial)
Query Warning Characters
Random Number
Seed Random Number
X Query work Space
Experimental Input
Experimental Output
Time
Set Time
Date
Set Date
Change Work Space
Query Work Space
Return Character Picture
Query current IO assignments
Set lobyte
Change Protection Class
Query Protection Class
Null Display mode
X Read Sector
X Write Sector
X Experimental User
Experimental Query Function
Experimental Change Function
TRS 80 Computer Functions

```

Expression formats, legend, syntax and conventions:

function,arguments,...,	Active Expression
\function,arguments,...,\	Neutral Expression
x,x1,...	"x" base numbers - f
d,d1,...	Decimal numbers - t
n,n1,...	"n" base numbers - vz
s0	prefixing string - v-,v+,v0
s,s1,...	character strings - vt,vf
Protection syntax - !,.../ (.....) <.....> <[.....]>	
Active syntax - S: \$fn,arguments/ - M: #fn,arguments;	
Neutral syntax - S: \$fn,arguments/ - M: #fn,arguments;	
<[.....]> partition [d], multi-partition [n], divider [']	
<[.....]> special condition encountered	
<[.....]> xxx not available	

