77-27

CMU COMPUTER SCIENCE SYSTEMS INTRODUCTORY USERS MANUAL

Editors: Jack Dills
Art Farley
Mary Shaw
Roy Levin
H. T. Kung

September 1972
January 1973
July 1973
August 1975
August 1976
August 1977

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

PREFACE

The following manual is intended to provide a usable introduction to computing on the CMU PDP-10. To accomplish this, a discussion of general computing procedures and the PDP-10 monitor is given, followed by descriptions of the available language systems. The manual does not provide full language descriptions (references are provided to necessary, useful language manuals); but through a short introduction, sample problems to try, and an annotated script, the manual hopes to impart to the user an introductory knowledge of what it is like, and what to expect, when using each of the discussed language systems on the CMU PDP-10.

Note that timely information can be found for many of the language systems in a printable text file <language>.DOC on the PDP-10. Information on which files are available can be found in DOC.DOC. To get a copy of a DOC file print SYS:<language>.DOC.

[illegible]
Sept. 1971

PREFACE TO THE 1976 EDITION

Major revisions have been done in this 1976 edition. The title of the manual is changed to "CMU Computer Science Systems Introductory Users Manual", since the manual now covers all the major systems available in the Department. Many people helped on this revision. Their names appear in the sections they wrote.

H. T. Kung
August 1976

TABLE OF CONTENTS

NOTE: The FORTRAN and BASIC language systems are fully described in the <u>DECSYSTEM10 Mathematical Languages Handbook</u>. Thus, no discussion of either is included.

I. PROCEDURES and MONITOR

H. D. Wactlar and C. D. Councill (eds.)
(Revised Aug., 1977)

1. Facility

1.1 Configuration

The Computer Science Department computing facility is presently (Sep 1977) composed of four general purpose time sharing systems and several stand-alone PDP-11 based research systems. The general purpose systems are CMUA, CMUB, and CMUD, all Digital Equipment Corporation PDP-10 systems; CMUA is a large KL10 running a CMU modified DEC 6.02VM monitor, CMUB and CMUD are KA10 processors both running a CMU modified DEC 5.06B monitor. The CMUB operating system also contains special code for the artificial intelligence (AI) devices attached to it. C.mmp is a PDP-11 based multiprocessor system of CMU design and implementation. C.mmp runs an operating system called HYDRA, again, a product of CMU research.

Most general purpose computing takes place on CMUA at present but some is also done on CMUD. CMUB is primarily dedicated to satisfying the stringent real-time requirements of the department's speech and vision research projects. C.mmp has sufficiently matured to the state of accepting general purpose time sharing users. By Fall 1977, it should offer a self contained ALGOL-68 subsystem to facilitate the user interface and programming tasks. Most of the systems are linked via the ARPA network to one another (as well as the entire network) for file and terminal transfer purposes.

Common terminal access is provided to all the CMU hosts via a PDP-11 based front-end system. Most terminals are concentrated in the third floor terminal rooms adjacent to the machine rooms. These several rooms contain all video terminals and are carpeted for maintenance of quiet working conditions.

1.2 Limited resources

Despite the apparent wealth of physical resources, computer cycles and on-line storage are very scarce commodities. The facility is intended to support the research and graduate education goals of the department's faculty, staff, and students. Use for any other purpose or transfer of access privileges to any other person is strictly prohibited. Users should carefully consider what projects they initiate and how they are implemented. When there are alternative approaches, take the time to learn them, and select the one which optimizes the man-machine utility. The facility can be a rich one only if we all cooperate.

2. Administrative Procedures

2.1 Usage Numbers

A computer usage application may be obtained from the PDP-10 operator. When completed, it may be returned to the Operations mailbox in the C. S. main office or to the PDP-10 operator. You will be notified either by phone or campus mail, probably within a week, as to your project-programmer number (PPN). It will contain eight alphanumeric characters. The first four characters are your project number; this is used for departmental accounting and statistics. The last four characters are your programmer number; this is made up of the initials of your first and last names with two digits appended. Your programmer number will be the first part of your DECtape labels and is sufficient to identify you in most cases.

2.2 Terminals

All terminals should be left with the power on, unless they are broken and awaiting repairs. Almost all terminals have an ONLINE switch or key; this should be in an online position (light on). Any terminal problems should be reported with a Trouble Report form, discussed below.

2.3 Trouble Report Forms

Hardware trouble report forms are located in a holder on the front wall of the terminal room, Science Hall 3103. These are to be filled out when you encounter hardware trouble with terminals. Both copies should be placed in the clear vinyl envelope on the terminal.

2.4 DECtapes and magtapes

DECtapes may be purchased in the CMU bookstore. Members of the Computer Science Department may borrow DECtapes for their use without charge by requesting them in writing by sending computer mail to GRIPE. If a user requires tapes for off-line storage, they should be requested with mail to GRIPE. Each DECtape is named with from five to ten alphanumeric characters. The first four characters will be your programmer number, with from one to six characters of your choice appended. MAGtapes will be named with a four digit number (which has meaning only to Operations) and you may have your man number appended to the label, if you so desire. This is the name which you will use when requesting that a tape be mounted, e.g., JS23-MYN. Computer Science owned tapes may not be removed from the machine room area.
The department is not responsible for the unreliability of non-departmental tapes.

3. Getting Connected to the System

3.1 The terminal Front-End

Most terminals are connected through the front-end system. Typing several ctrl-C's when the terminal is connected only to the front-end and not yet through to a host, will cause it to recognize your terminal speed and prompt for the host of your choice. Respond with the appropriate unique one-character identifier:

A - for CMUA
B - for CMUB
C - for C.mmp
D - CMUD
H - for help message
* - for Cm* Host
S - AI-Speech
V - AI-Vision
L - Line Info
U - Uptime

If the host you specify is unavailable, the front-end should so indicate. Once connected through to a host, you must login or the host will timeout your connection and throw your terminal back to the front-end state.

You may return to the front-end to initiate a connection to another CMU host at any time by transmitting the escape character (octal code 37, ctrl-leftarrow or ctrl-underscore on most CMU terminals and hosts). Your jobs remain on the host from which you escaped and you may reconnect to them at any time. When you logout from a particular host, your terminal is automatically forced back to the front-end whether or not you type the escape-back character.

3.2 Telephone lines

In addition to the hard wired terminals, users may dial into the various systems over 10, 15 and 30 character per second lines. Consult LINES.HLP on the PDP-10 systems for the latest list of phone numbers. At the moment, the list is as follows:

Front-End lines [Variable Baud rate]   621-3526 (10-line rotary)

Type ctrl-C's until the Front-End responds with a host name request. It determines the baud rate by examination of the bits in the ↑C chars. Answer with the unique one character identifier for the host that you wish to connect to.

Individual-machine dialup ports. These do not connect to the front-end, but go directly into the indicated machine.

110 Baud (Teletype)		
CMUB	621-3525	TTY0
134.5 Baud (Datel, 2741)		
CMUD	683-8330	TTY32
	683-8331	TTY33
300 Baud (TI700, CRT's, etc.)		
CMUD	621-3520	TTY37
	621-3521	TTY44
	621-3522	TTY43
CMUB	621-3524	TTY5
	683-8333	TTY4

110-baud dialups assume TELETYpe as the terminal type, while 300-baud dialups assume TI. Use SET TTY <x>, where <x> is ANNARBor,BEEHIVe, LA36 (DECwriter), et cetera. SET TTY must be typed after logging in.

The operator's outside line is 412-682-7086.

Other useful telephone numbers:

CMU Campus	621-2600
Computer Science Office	x141
Computer Science Lounge	x144
PDP-10 machine room	x176
CMUA and C.mmp rm	x131
Main terminal rm (3103)	x129
A.I. terminal rm	x169
	x153
C.S. Engineering lab	x170

4. Using CMUA/B/D

4.1 Logging in to the host

The LOGIN command is used to gain access to the system or to start a new job. The dialogue is as follows:

```
.LOGIN <PPN>
Job # <CMU monitor name>
Password:
time   date
<message of the day>
```

where PPN is your usage number (e.g. C410HB00). After 'Password:' type your password, which doesn't echo. If you wish to change your password, type an altmode instead of a carriage return and the system will prompt you for your new password.

When you are finished using the system and wish to logout, use the KJOB command (described under Useful Monitor Commands).

4.2 Communicating with the operator

4.2.1 The PLEASE command

PLEASE is a monitor command which puts the issuing terminal, and eventually, the CTY (console teletype) into a special communications mode. This mode may be evoked when logged in and in monitor mode, by typing:

```
.PLEASE <text>
```

You only need to type the word "please" once. If the CTY has a job on it, or running SYSTAT, or in another PLEASE, the message:

```
OPERATOR BUSY, PLEASE HANG ON
```

will print on your terminal. You can terminate the PLEASE with a control-C or wait until the CTY is free. When it is free, your terminal will print:

```
OPERATOR HAS BEEN NOTIFIED
```

and then your message, along with indentifying information about you and several bells, will print on the CTY. Now both terminals are in "please mode". Any line typed on either terminal, terminated by a <CR>, will print out on the other terminal and will otherwise be ignored by the system. Thus a two-way communication is established.

If you have more than one line of text to communicate to the operator, end the first line with three periods (...). When you are finished with your message and wish to receive the operator's reply, end your message with the letters GA (go ahead). When you wish to completely terminate your end of the interaction, end your last sentence with GA or BYE. This gives the operator the chance to add one last thing, if he should need to. "Please mode" may be terminated by an altmode or a control-C on either terminal. However, it is courteous to wait until the operator altmodes out of the PLEASE. Both terminals will then be in monitor mode. The most frequent use of PLEASE is to request the mounting and dismounting of tapes and disks.

4.2.2 The SEND command

The SEND command is used to send a one-line message to another system user. The target user may be specified by his job number or by his terminal number. The operator is either CTY or OPR. Examples:

```
SEND TTY62:  <text>
SEND JOB 17  <text>
SEND OPR:    <text>
```

SEND leaves the issuing user in monitor mode and the receiving user at the same place he was before the SEND. The format of the message on the receiving terminal is:

```
;;13:40:26 TTY43 (sender name): <text>
```

The proper way to converse with the operator is with PLEASE.

4.2.3 GRIPE

CMU does not use the system program provided by DEC, called GRIPE, for "griping" purposes. Instead, we have an account with the name GRIPE and when you wish to GRIPE you send mail (using the MAIL program) to that account. GRIPEing is the word used for almost any type of user request that doesn't require an immediate response. ("immediate" means minutes or hours.) The following types of requests should be mailed to GRIPE: requests for accounts, tapes, disk storage increases, name changes and the like, as well as questions and suggestions about system cusps and the operating system. The GRIPE mail.box is then read later by the Operations staff and you will generally receive an answer to your problem/question by computer mail, unless you specify otherwise.

4.2.4 Getting tapes mounted

DECtape drives are available only on the CMUB and CMUD. If your account is on the CMUA and you wish to use a DECtape, then use the cusp called, TOOK. Type HELP TOOK on the CMUA for more information. If you are going to use DECtapes on either the CMUB or CMUD, then follow the instructions below. The first thing to do is to get a unit assigned for your tape.

```
Type:           .ASSIGN DTA (for DECtape)   or
                .ASSIGN MTA (for magtape)
```

The monitor will respond with:

```
                DTA2 ASSIGNED
```

or, if no unit is available, it will respond:

```
                ?ASSIGNED TO JOB #
```

After a unit is assigned to you, you will notify the operator to mount your tape by using the monitor command PLEASE. In your request specify the tape name, tape unit, and whether the tape should be enabled for writing. If you do not specify "write enabled" (W/E), the operator will write lock the tape. Remain in PLEASE mode until the operator responds to your request. He may say:

```
                NN00ABC MOUNTED ON DTA2 ENABLED
```

or, since the monitor recognizes 6 DECtape units and 3 MAGtape units, and we have only five DECtape drives and two MAGtape drives, there may not be a drive free for you even though you have a unit assigned. If this is the case, the operator will try to get a drive for you as soon as possible. The drives are allotted on a first-come-first-served (FCFS) basis. If you need a drive urgently or only for a minute, use systat to find out who are using the drives and contact them with "SEND" to ask them if they need the drive. If you do not need a drive immediately, the operator will mount your tape and notify you when a drive is free.

The tape drives are very much in demand, so please be considerate of others. When you finish with a tape, be sure to tell the operator to dismount it immediately, and deassign the DTA thus freeing the drive for someone else. You can type:

```
                .DEASSIGN DTA2
```

to make the unit available for others. If you are using the same unit number for more than one tape, be sure to reassign the unit between tapes by again typing:

```
                .ASSIGN DTA2
```

and so a fresh copy of the directory will be read into core and you will not be using the directory from the last tape.

If you are logging off, the unit will be returned to the pool, and your tape will be dismounted automatically if you have forgotten to deassign the drive.

4.3 Line Printer Usage

There are three printers in the department, two dual case and one upper-case. One of the dual case printers is connected to C.mmp, and the other is connected to the CMUA. The upper case printer is switched between the CMUB and the CMUD.

The dual case printer on the CMUA is located near the entrance from the machine room to the 3rd floor terminal room. The upper case line printer is located near the front doors to the machine room along the aisle.

When you pick up your listing, it may be attached to listings done before and after it. It is considerate to separate all the listings attached to yours and place other people's listings on the table near the printer. This process is called "bursting".

Each listing is preceded by a large page with the hame of the file, your programmer number, and miscellaneous dates and times. The listing is also followed by a similar page.. You can tell them apart because one says BEGIN at the four corners and one says END. Both pages will have a decimal number at the extreme edges of the middle line. Both pages have the same number; you can use this to find the limits of a listing you are trying to burst.

There are boxes beside all printers for old listings and break pages. The paper gets recycled, and the money is used for various departmental festivities. Please be neat about putting paper in the box.

4.4 File Space Allocation

Every account is allocated a quota of a modest amount of public storage disk space. Users are expected to live within this quota. Should it prove inadequate for pursuing a serious line of research, reasonable requests for larger quotas will be considered when submitted via mail to GRIPE with an adequate explanation.

There are three public storage disks on the CMUA: DSKB, DSKC, and TEMP. On the CMUB there are three: DSKA, DSKC, and TEMP. And on the CMUD, there are four: DSKA, DSKB, DSKC, and TEMP. DSKA: on both the CMUB and CMUD is the swapping drum and may be utilized only for temporary user files only while the user is logged in. We are always running out of space on DSKB and DSKC, and often run out of space on TEMP. About once a week a program is run which deletes everything from TEMP that you haven't looked at that week. TEMP is intended to hold intermediate files and other things that have a short lifespan. It is considered socially unacceptable to try to store things there permanently and to request restorations of files that have been deleted by the "disk purger".

4.5 Terminal Control

On the terminal, there is a special key marked CTRL called the Control Key. If this key is held down and a character key is depressed, the terminal types what is known as a Control Character rather than the character printed on that key. In this way, more characters can be used than there are keys on the keyboard. Most of the control characters do not print on the terminal, but cause special functions to occur, as described in the following sections.

There are several other special keys that are recognized by the system. The system "sees" all of the characters typed into your terminal and, most of the time, sends the characters to the program being executed. The important characters not passed to the program are also explained in the following sections.

The ALTMODE or ESCAPE key

The ALTMODE key ($), which is labeled ALTMODE, ESCAPE or PREFIX, is used as a command terminator for several programs and the monitor. Since ALTMODE is a non-printing character, the terminal prints ALTMODE as a dollar sign.

Control-C

Control-C (↑C) interrupts the program that is currently running and takes you back to the monitor. The monitor responds to a control-C by typing a period (.) on your terminal. You may then type another monitor command. For example, suppose you are running a program in ALGOL and you now decide that you want to leave ALGOL and run a program in MACRO. When ALGOL requests input from your terminal by typing an asterisk (*), type control-C to terminate ALGOL and return to the monitor. You may now issue a command to initialize MACRO (R MACRO). If the program is not requesting input from your terminal (i.e. the program is in the middle of execution), when you type control-C the program is not stopped immediately. In this case, type control-C twice in a row to stop the execution of the program and return control to the monitor. If you wish to continue at the same place that the program was interrupted type the monitor command CONTINUE. Or if you wish to start the same program over again, type the monitor command START.

Control-H

Control-H (↑H) performs a function similar to RUBOUT but does not echo the deleted character or type any backslashes. On video terminals, the deleted character is erased from the screen and the cursor positioned in the empty space. Thus, the screen appears as though the erroneous character had never been typed.

Control-O

Control-O (↑O) tells the computer to cancel terminal output. For example, if you issue a command to type out 100 lines of text and then decide that you do not want the typeout, type control-O to eliminate the output. Another command may then be typed as if the typeout had terminated normally.

Control-Q

Control-Q (↑Q) resumes the terminal output halted by the control-S (↑S).

Control-R

This command (↑R) will cause the system to retype all characters that are presently in the current line input buffer. This is very handy when several corrections to a line have been typed and you are uncertain as to what your input line to the machine really is. This only works before you have typed a carriage return. (Implemented on the CMUA only, Sep 1977)

The RETURN key

Pressing this key causes two operations to be performed: (1) a carriage return and (2) an automatic line-feed. This means that the typing element returns to the beginning of the line (carriage return) and that the paper is advanced one line (line-feed). Commands to most programs and the monitor are terminated by pressing this key.

The RUBOUT key

The RUBOUT key permits the correction of typing errors. Depressing this key once causes the last character typed to be deleted. Depressing RUBOUT n times causes the last n characters typed to be deleted. RUBOUT does not delete characters beyond the previous carriage return, line-feed or altmode. Nor does RUBOUT function if the program has already processed the character you wish to delete.

The monitor types the deleted characters, delimited by backslashes. For example, if you were typing CREATE and got as far as CRAT, you can correct the error by typing two RUBOUTs and then the correct letters. The typeout would look like:

```
CRAT\TA\EATE
```

Notice that you typed only two RUBOUTs, but \TA\ was printed. This shows the deleted characters, but in reverse order.

Control-S

Control-S (↑S) halts terminal output in progress. Output is resumed by typing a formfeed (↑L) or a ↑Q which are not passed to the user program, or by ↑S which is passed (thus, ↑S↑S sends one ↑S). The state is also reset by ↑O and ↑C↑C. It is especially useful on video terminals so that the output does not scroll off the screen before it has been read.

Control-T

Typing control-T (↑T) at your terminal will give a one line status of the running job with out interrupting it. It is handy if you are in the middle of running a long program and you want a "systat" of it, but don't want to log in another job or destroy your core image. (Implemented on the CMUA only, Sep 1977)

Control-U

Control-U (↑U) is used if you have completely mistyped the current line and wish to start over again. Once you type a carriage return (<CR>), the command is read by the computer, and line-editing features can no longer be used on that line. Control-U causes deletion of the entire line, back to the last carriage return, line-feed or atlmode. The system responds with a carriage return, line-feed so that you may start again.

Control-Z

This is an end of file character when the input device is a terminal (TTY), similar to an end of file mark on a magtape.

Modifying terminal characteristics

When you login to the system the terminal (TTY) characteristics are defaulted to the appropriate set for that terminal. If you wish to modify them, there is a TTY command which declares special properties of the terminal line to the scanner or Front End service. The command format is:

```
TTY NO word
```

NO = the argument that determines whether a state is to be set or cleared. This argument is also optional.

word:= the various words representing states that may be modified by this command. Some of the words are as follows:

TTY UC	The monitor translates lower-case characters to upper case as they are received.
TTY LC	The translation of lower-case to upper-case is suppressed.
TTY WIDTH n	The carriage width (the point at which a free carriage return is inserted) is set to n.
TTY NO CRLF	The carriage return normally output at the end of a line exceeding the carriage width is suppressed.
TTY NO BLANK	Eliminates blank lines from the output. Increases the data density on screens and thermal paper.
TTY PAGE	One full screen's worth of data is output on to the video terminal and then stopped.

	Typing any character resumes output with another "page".
TTY NO BACKSP	Disables CTRL-H's delete character function. CTRL-H gets passed to the program (for APL on Graphics terminals).
TTY <type>	where <type> = INF|BEE|TELE|GRAPHIC|; sets all of the characteristics to the appropriate set for the specified terminal type.
TTY RESET	Resets all the characteristics back to the default for this line.

4.6 Useful Monitor Commands

Commonly used monitor commands (unique abbreviations are acceptable) :

ASSIGN

<physical device> <logical name>
allocates an I/O device (DECtape, magtape, disk) to the user's job and optionally assigns a logical name designated by the user to that device. Example:

```
.ASSIGN DTA DT11
DTA3 assigned
```

ATTACH

<job number>[project-programmer number]
detaches the current job, if any, and connects the console to a detached job. The user is prompted for the appropriate password, as in LOGIN, before the ATTACH is completed.

CLOSE

<dev:>
terminates any I/O currently in progress on the specified device.

COMPIL

<list of source file names separated by commas>
produces relocatable binary files for the specified program(s) by calling the appropriate compiler as determined by the source file name extention (ALG for ALGOL, MAC for MACRO, F4 for FORTRAN, BLI for BLISS, SAI for SAIL). Example:

```
.COMPILE TEST.MAC
```

CONTINUE

starts the program at the saved program counter (PC) address stored by a halt (↑C) command.

COPY

<dev: file name> = <dev: file name>
transfers files from one standard I/O device to another.

CREATE

<file name>
calls LINED (SOS) to create a new file. Example:

```
.CREATE TEST.MAC
```

DAYTIME

types the date followed by the time of day.

DDT

saves the program counter (PC) and starts the program at the dynamic debugging module optionally loaded with the compiled program.

DEASSIGN	<logical or physical device name> returns the I/O device to the system's available pool. Example: `.DEASSIGN DTA3`
DEBUG	<list of file names separated by commas> performs the compile and loading functions and in addition loads DDT which it enters on completion of loading. Example: `.DEBUG TEST.MAC, TEST2.F4`
DELETE	<list of file names or groups separated by commas> automatically runs PIP to delete the specified files. Example: `.DELETE TEST.MAC,*.REL`
DETACH	disconnects the terminal from the user's job without affecting its status. The terminal is now free to control another job.
DIRECT	<dev: file name> lists specified directory entries. Some options are:

DIR	List name, length, creation. date, etc. for each file in your disk area.
DIR/Fast	Same as above, but lists only file names.
DIR DTA3:	List all the files on your DECtape on DTA3:
DIR TEXT7.MAC	Print the vital statistics of file TEXT7.MAC on your disk area
DIR/since:-3d:4:27	Print info for all files created since 3 days, 4 hours, 27 minutes ago
DIR/since:Wed	Print info for all files created since last Wednesday
DIR/before:25-Oct-73	Print info for files created before 25-Oct-76
DIR/access/since:Wed	Print info about files accessed since last Wednesday

DIRECT outputs files by allocation unless /BLOCKS is specified.

File names and extentions may be specified with wild cards (* and ?). The wild cards work as follows: "?" matches any character or null and "*" matches any file name or extention.

DIRECT will accept CMU PPNs with "*" and "?" wild cards. It will also accept DEC PPNs. Use [**] for a completely wild PPN.

DSK	<job number> types the incremental and total disk usage in 128 word blocks read and written.
EDIT	<file name> calls LINED (SOS) to edit an already existing file. Example: .EDIT TEST.MAC
EXECUTE	<list of file names separated by commas> performs the compiling and loading functions and initiates program execution. Example: .EXECUTE TEST.MAC
GET	<file name> loads a previously saved core image but does not begin execution.
HALT	stops the job and saves the program counter (PC). Bypasses control-C trapping.
HELP	<name> prints helpful documentation for various commands and programs.
KJOB	<arg> initiates the logoff sequence. In response to CONFIRM:, type one of: BDFHIKLPQSUWX B to perform algorithm to get under quota D to delete all files (asks Are you sure?, type Y or <CR>) F try fast logout by leaving all files on DSK H type this text I to individually determine what to do with all files after each file name is typed out, type one of: EKPQS, or H to get more help K to delete all unpreserved files L to list all files P to preserve all except TMP files Q to report if over quota S to save all except TMP files U same as I but automatically preserve files that are already preserved W to list files when deleted X to suppress listing deleted files (default) If a letter is followed by a space and a list of File Structures, only those listed will be affected by the command. CONFIRM: will be typed again. A file is preserved if the access protection is >100.
LABEL	DTAn:<label name> writes a name logically onto a DECtape. n is the drive number and the label may be any 6 characters.

LIST	<line printer>=<file name specifier>/switch queues files to be printed on the line printer. Switches include:

#	(# = 1 - 9) print multiple copies
D	delete after printing (default for all .LST, .TMP and .CRF files)
P	preserve after printing

LOAD	<list of file names separated by commas> performs the compiling and loading functions to execute the core image of a runnable program.
LOGIN	<project-programmer number> initiates the login sequence.
MAIL	<name list>/<switch:arg>/<switch:arg>...../<switch:arg>

sends general messages to other users for them to read when they next LOGIN. The basic format is shown above.

```
<namelist>:= <mailee> \ <mailee> + <namelist>

<mailee> := name \ programmer # \ PPN \ entire account

name := lastname \ firstname lastname \ lastname,
        firstname \ name@host name

<switch>:= CC:arg \ DELMAIL \ DISTRIBUTION:arg \ EXPAND \
        FILE:arg \IDENT:arg \ LINENUMBERS \ NOWHO \
        RETRY \ SAVMAIL \ SUBJECT:arg \ TRANSCRIBE
```

MAIL also accepts command files using the '@<file>' construct.

All mail is appended to a file called MAIL.BOX on the object user's disk area, and may be read with RDMAIL or with any editor (e.g. LINED (SOS) or TECO).

For more information, see DOC:MAIL.DOC.

MAKE	<file name> calls TECO to create a new file.
MOUNT	<disk structure name> establishes a user file directory on a dismountable file structure. Also used for gaining access to TEMP:.
PJOB	types host, job number and project-programmer number of job running on the terminal on which this command is typed.
PROTECT	<dev:><file specifier><file protection code> alters the access protection associated with the specified files. The access protection is indicated

by three digits. Each digit represents a particular class of user. The first digit represents the owner of the file, the second represents user with the same project number of the owner, and the third represents all of the other users. Each number in the three digit code can be one of the following:

7	No access privileges
6	Execute the file
5	Read and execute the file
4	Append, read, and execute the file
3	Update, append, read, and execute the file
2	Write, update, append, read and execute the file
1	Rename, write, update, append, read, delete and execute the file
0	Change protection, rename, write, update, append, read, delete and execute the file

The standard default protection is <055>.

R <system program name>
executes the named program residing on the system area (SYS:). Example:

```
.R FILCOM
```

RDMAIL manipulates computer mail.
It provides a powerful set of facilities for examining and manipulating messages, sending mail, and archiving old mail. The most commonly used commands appear below. For full details see DOC:(or SYS:)RDMAIL.DOC or type DOC when in RDMAIL.

Headers (optional <msg-seq>) means type a one line indication of each of the messages in <msg-seq> (defaults to all of the new messages).
DELete <msg-seq> means mark each of the messages as deleted.
Type <msg-seq> means type each of the messages on the terminal.
<line feed> means type the next message.
List <msg-seq> means list each of the messages to a file. It will prompt for a file name which defaults to LPT:MAIL.
SEnd <person> means send a piece of mail to <person>. You will be prompted for subject, etc.
Answer <msg-num> means answer a message. Similar to SEND, but with additional defaults.
Exit means return to monitor mode, deleting messages so marked.
? means quick summary of command names and special functions.

Message numbers range from 1 to the highest numbered message that exists in the file. A msg-seq is a set of messages. For example: 4,6:9,.:15,* means messages 4,6,7,8,9,14,15 and 23 assuming that the "current" message is 14 and the last one in the file is 23.

READ

<file name>
calls LINED (SOS) to examine an already existing file with line number printout turned off.

LINED prompts with a "*"; most commands should end with a carriage return <cr>.

The most commonly used commands for READ mode are:

H	Give HELP. (Note: all modify commands illegal READ mode)
<LF>	A line-feed print the next line.
<ALT>	An escape or altmode ($) prints the previous line.
B	Go back to the BEGINNING of the file (does not print anything).
B/n	Go to the BEGINNING of page n (e.g., B/3).
P.	Type the current line.
P/.	Type the current page.
P/.+1	Type the next page.
P.:*	Type from the current line through the end of the page.
P	Type the current line and the next 15 lines.
O	Type a WINDOW around the current line: +/- 5 lines.
F<string><esc>	FIND the first occurence (between the current line and the end of the file) of <string>. The <CR> is needed.
F	FIND, use the <string> of the last F command given.
=.	Show the <line-number>/<page-number> of the current line.
←SEE=1	Enable line-number printing.
←SEE=0	Disable line-number printing.
=File	Show the name of the file currently being READ.
E	END reading and return to the monitor.

For more information, use the H command or see the SOS manual.

RENAME

<dev: new file name> = <dev: old file name>
runs PIP to change a file name. Example:

```
        .RENAME TEST1.MAC = TEST.MAC
        FILES RENAMED:
        TEST.MAC
```

RUN

<file name>
runs the core image previously loaded and SAVE'd with that file name. Example:

```
        .RUN TEST.SAV
```

SAVE <file name>
copies the core image currently loaded in core onto the specified file so that it may be RUN at a later time. Example:

```
.LOAD TEST.MAC
.SAVE TEST.MAC
Job saved
```

SSAVE <file name>
same as SAVE, but it saves the core image in a form to be shared by multiple users when running.

START starts the execution of an already loaded core image.

SYSTAT <arg>
provides status information about the system and about jobs which are running. The complete SYSTAT printout is much longer than most users want to see; more frequently, some subset of the SYSTAT printout is is of interest. SYSTAT may be abbreviated as SY, SYS, SYST, etc. For a more complete help text, type SYS H. These are a few of the more frequently performed operations:

SYS J	Job status: print status of all jobs
SYS W	WHO: identifies system users by name
SYS <n>	status for one job: print SYSTAT information for job <n>
SYS S	Short: print an abbreviated version of SYS J
SYS F	Print summary of available file space
SYS Q	Show the line-printer spool queue.
SYS #<n>	status for TTY: print SYSTAT info. for jobs controlled by TTY line number <n>
SYS "string"	Selective print: edits full SYSTAT, but prints those lines containing the specified string.
SYS H	Print SYSTAT's internal help text.
SYS W H	Print WHO's internal help text.

TECO <file name>
calls TECO to edit and already existing file.

TIME <job number>
causes typeout of total runtime since the last TIME command, total runtime since LOGIN, and and integrated product of runtime and core size.

ZERO <dev:>
clears the directory on the device specified.

4.7. COMPILE command switches

The COMPIL, LOAD, EXECUTE and DEBUG commands may be modified by a variety of switches. Each switch is preceded by a slash and is terminated by a non-alphanumeric character, usually a space or a comma. An abbreviation may be used if it uniquely identifies a particular switch.

These switches may be either temporary or permanent. A temporary switch is appended to the end of the filename, without an intervening space, and applies only to that file. Example:

```
.COMPILE A,B/MACRO,C
```

(The MACRO assembler applies only to file B.) A permanent switch is set off from file names by spaces, commas, or any combination of the two. It applies to all the following files unless modified by a subsequent switch. Example:

```
.COMPILE /MACRO A,B,C
.COMPILE A /MACRO B,C
.COMPILE A,/MACRO,B,C
.COMPILE A,/MACRO B,C
```

Compilation listings - Listing files may be generated by switches. The listings may be of the ordinary or cross reference type. The operation of the switch produces a disk file with the extention .LST, queues it, prints it, and then deletes it. The COMPILE switches LIST and NOLIST cause listing and nonlisting of programs and may be used as temporary or permanent switches. Listings of all three programs are generated by:

```
.COMPILE /LIST A,B,C
```

A listing only of program A is generated by:

```
.COMPILE A/LIST,B,C
```

Listings of programs A and C are generated by:

```
.COMPILE /LIST A,B/NOLIST,C
```

The COMPILE switch CREF is like LIST, except that a cross reference listing is generated (FILE.CRF), processed later by the CREF CUSP which generates the .LST file, queues, prints and deletes it. Unless the /LIST or /CREF is specified, no listing file will be generated.

Since the LIST, NOLIST and CREF switches are commonly used the switches L, N and C are defined with the corresponding meanings, although there are (for instance) other switches beginning with the letter L. Thus, the following command:

```
.COMPILE /L A
```

produces a listing file A.LST (and A.REL).

Standard Processor - The standard processor is used to compile or assemble programs that do not have the extentions .MAC, .CBL, .F4, or .REL. A variety of switches set the standard processor. If all source files are kept with the appropriate extentions, this subject can be disregarded. If the following command:

```
.COMPILE A
```

is executed and there is a file named A.(that is, with a null extention), the A. will be translated into A.REL by the standard processor. Similiarly, if the following command:

```
.COMPILE FILE.NEW
```

is executed, the extention .NEW although meaningful to the user, does not specify a language; therefore, the standard processor is used. The user must be able to control the setting of the standard processor, which is FORTRAN IV, at the beginning of each command string.

Forced compilation - Compilation (or assembly) occurs if the source file is at least as recent as the relocatable binary file. The creation time for files is kept to the nearest minute. Therefore, it is possible for an unnecessary compilation to occur. If the binary is newer than the source, the translation does not usually have to be performed.

There are cases, however, where such extra translation may be desirable (i.e. when a listing of the assembly is desired). To force such an assembly, the switch /COMPILE is provided, in temporary and permanent form. For example:

```
.COMPILE /CREF/COMPILE A,B,C
```

will create cross reference listing files A.CRF, B.CRF and C.CRF, although current .REL files may exist. The binary files will also be recreated.

4.8. Useful Utilities

4.8.1 PIP - Peripheral Interchange Program

```
<destination specification>←<source specification>
```

PIP is a basic systems program of the PDP-10 which provides the user with the necessary facilities for handling existing data files. Actions possible among others, are transferring files from one standard I/O device to another standard I/O device, listing and deleting directories, simple editing, changing protection codes, and controlling magnetic tape functions. Note that PIP will not assign devices, thus any device assignments must be previously done. All specifications are in the following form:

```
<dev:><file.ext>,<file.ext>,...<file.ext>[PPN]/<switch>
```

PIP will accept wild cards (? and *) for both file names and PPN's. It will also accept octal numbers; a number sign or hash (#) is used as a flag to indicate the presence of an octal constant in a file name or extention. PIP will accept command files in the following format: FILE.EXT@, where the default extention is .CCL. Below is a summary of PIP switches in alphabetical order.

A	line blocking
B	binary processing (mode)
C	suppress trailing spaces, convert multiple spaces to tabs
D	delete file
E	treat (card) columns 73-80 as spaces
F	list disk directory (file names and ext. only)
G	ignore I/O errors
H	image binary processing (mode)
I	image processing (mode)
L	list directory
M	see MTA switches below
N	delete sequence numbers
O	same as /S switch, except increment is by 1
P	FORTRAN output assumed. convert format control characters for LPT listing /B/P copy FORTRAN binary
Q	print (this) list of switches and meanings
R	rename file
S	resequence, or add sequence numbers to file; increment is by 10
T	suppress trailing spaces only
U	copy block 0 (dta)
V	match parentheses (<>)
W	convert tabs to multiple spaces
X	copy specified files
Z	zero out directory

MTA SWITCHES:
Enclose in parentheses ().

M followed by	8 means select	800 B.P.I. density
	5	556
	2	200
	E	even parity
	A	advance MTA 1 file
	D	advance MTA 1 record
	B	backspace MTA 1 file
	P	backspace MTA 1 record
	W	rewind MTA
	T	skip to logical EOT
	U	rewind and unload MTA
	F	mark EOF

(M#NA),(M#NB),(M#ND),(M#NP) mean advance or backspace MTA N files or records.

4.8.2 FILCOM - File comparision

```
<output specifier>=<input specifier>
```

FILCOM compares two files in either ASCII mode or binary depending upon switches or file name extensions. All standard binary extensions are recognized as binary by default. Switches are listed below in alphabetical order:

```
/A compare in ASCII mode
/B allow compare of Blank lines
/C ignore Comments and spacing
/S ignore Spacing
/H type this Help text
/#L Lower limit for partial compare
       or number of Lines to be matched
       ( # represents an octal number)
/#U Upper limit for partial compare
/Q quick compare only, give error message if files differ
/U compare in ASCII Update mode
/W compare in Word mode but don't expand files
/X expand files before word mode compare
```

4.8.3 LPT128 - Full ASCII line printer output

LPT128 prints files which use the entire 128-character ASCII set. The user has the option of printing special characters with or without question marks preceding them (SOS input format). If question marks are not printed, then the special character equivalent is printed without the question mark, e.g. "?*" would print as "*", not

.

The default mode has all the "right" options for printing SOS files. To select other options, type altmode to the request for a file. To any question reply either Y, N, carriage return or ?. ? will give a slightly longer explanation of the question and then ask it again. If a reply is terminated by an altmode no further questions will be asked. If no reply is given, i.e., just a bare carriage return, then the value is not changed from its previous setting.

A command file (containing filenames as well as mode setting commands) may be specified by "@filename". Lines from the command file will be typed as they are read.

LPT128 has a "SPY" option. In setting the modes, you may select a positive value for the SPY size. LPT128 will then print only that many lines from each file. Typical use is to specify wildcards in the command file and see the first few lines of each file. Note that when operated in SPY mode, LPT128 will not update the access date of any files examined.

4.8.4 QUOLST - Listing your disk quotas

QUOLST is a program which informs you how much disk space you have used and how much space you have left. It also lists the amount which the system has left. To run it, simply type:

```
        .R QUOLST
```

It has no switches. It gives the user's project-programmer number. For each structure in your search list, it gives the structure name, the number of blocks used, and the amount of space left in each of three categories:

(IN)	logged in (FCFS) quota
(OUT)	logged out quota
(SYS)	to all users on the system

4.8.5 MOVE - Moving files easily

The MOVE program allows a user to transfer files from any disk and/or PPN to any disk and/or PPN (provided that the user has the proper rights, of course). The files are deleted from the source structure/PPN after they have been transfered to the destination structure/PPN. MOVE leaves access and creation dates unchanged on all files it touches. In addition, the program can be used to delete line numbers from files (but without modifying the creation or access date of the file -- this is especially useful for shrinking files to save disk space, but without confusing the file's "age".)

The program prompts the user for the following information:

FROM PPN: The source PPN. Defaults to the current job's PPN.

FROM STRUCTURE: The source structure (a disk). Default is DSK. You MUST specify a DISK STRUCTURE (e.g., DSKB, but not DTA1).

TO PPN: The destination PPN. Defaults to the source PPN.

TO STRUCTURE: The destination structure. This defaults to the source structure.

FILES: The file(s) that are going to be moved. You can specify a list of files, using a "," as separator. The standard "*" and "?" conventions are allowed in the filename.ext specifications. If the switch /D (decide) is given following a file name, that file and all following files on that line are then prompted and the user is given a chance to decide if that file is really to be transferred. (Typing N to the prompt skips that file, anything else transfers it.)

NOTE: The source or destination structure can be specified as DSK, in which case the search list associated with the LOGIN PPN is used to access and enter files. You can use DSK as a structure specification only if the destination and source PPN's are different or if the other structure is not in the search list.

4.8.6 SETSRC - Modifying your search list

This program is used to alter the succession of disk areas and file structures searched when a file is requested. The switches for SETSRC are listed below:

```
Abrev.: F.S.=file structure, S.L.=search list

User S.L. Commands:

T        Type S.L.
C list   Create new S.L.
M list   Modify F.S. parameters of current S.L.
A list   Add F.S. to current S.L.
R list   Remove F.S. from current S.L.

        list=   /sl str/fs/fs, str/fs/fs, ... ,str/fs/fs /sl
        str=    F.S. name (e.g. DSKA:)
                or '*' to indicate all F.S. in current S.L.
        /fs=     F.S. switches:
                /WRITE  Write enable F.S.
                /NOWRITE Write lock F.S. (this job only)
                /CREATE Allow file creation on F.S. when DSK: specified
                /NOCREATE Allow file creation only if F.S. named
                /W=/WRITE /R=/NOWRITE /C=/CREATE /N=/NOCREATE
                DEFAULT (NO SWITCHES)= /WRITE/CREATE
        /SL=    OPTIONAL S.L. SWITCHES:
                /SYS    Add auto. search of SYS: to DSK: specification
                /NOSYS  Remove ...
                /NEW    Add auto. search of [1,5] to SYS: specification
                /NONEW  Remove ...
                /LIB:[PROJ,PROG] Add auto. search of library [proj,prog]
                        to DSK: specification.
                /NOLIB  remove ...
                Default (no switches) leave DSK: specification as is


SYS=M/SYS
NOSYS=M/NOSYS
etc. for NEW, NONEW, LIB[proj,prog], and NOLIB.

System S.L. Commands:
        All commands except TS require user logged in as [1,2].

TS       Type system S.L.
CS list Create new ...
MS list Modify F.S. parameters in ...
AS list Add F.S. to ...
RS list Remove F.S. from ...

        list=   same as for user S.L. except S.L. swithes not allowed.
```

Path Commands:

```
TP              type default path
CP path Create new default path
SCAN            enable scanning
NOSCAN          desable ...

        path=   [proj,prog,sfd,...,sfd] /s
        proj,prog= UFD name
        sfd=    subfile directory names
        /s=     switches:
                /SCAN   Enable scanning
                /NOSCAN Disable ...
                /S=/SCAN /N=/NOSCAN
                default (no switches) leaves scanning as is
```

4.9 The CMU BATCH System

NOTE: The CMU BATCH system is only run, at present (Sep 1977), on the CMUA and CMUD. To obtain this BATCH system on the CMUD, you must run CMUQ from SYS:; the default BATCH for the CMUD is DEC's BATCON.

BATCH is the CMU mini-batch system. It reads .CTL files which contain teletype input scripts and writes .LOG files which contain the full terminal session resulting from executing the commands in the .CTL file. Place characters into the input file exactly as desired. For example, in order to get a ↑C in a file use the SOS ?# or TECO 3I$ constructs. The ↑C will not be passed to the job until it requests input from the terminal. The LOGIN and KJOB sequences are provided by the mini-batch monitor. Do not include them in the input file! BATCH provides two job streams, with different scheduling policies. Each stream selects jobs to run from the same set of requests. One stream (stream 2, by chance) is for short jobs (runtime not more than three minutes) only, and uses a strict FIFO discipline. Jobs run in this stream will not be put into low priority, but will run in the normal HPQ 0. The second stream (stream 1) uses a very primitive scheduling algorithm that works fairly well in our environment. At any given time the jobs queued up for batch processing fall into two categories: those whose startup time has passed (runnable) and those whose startup time has not been reached (waiting). The scheduler is concerned only with runnable jobs. The scheduling algorithm changes with the time of day.

0200 - 0700	FIFO
0700 - 0200	Shortest job first (wraparound thru midnight)

Additionally, in shortest job first mode, any job that requests more than 10 minutes of CPU time will not be started if the system is loaded (an arbitrary threshold meaning more than 34 jobs logged in or an active swapping ratio greater than 0.9). If the shortest runnable job requests more than 10 minutes, no job will be started and BATCH will check the load again in 15 minutes. The system must remain in an unloaded state for 30 minutes before a long job will be started.

Once a job is started it is run to completion. For instance, a job that requests 25 minutes of CPU will not be stopped because another job that requests 2 minutes becomes runnable after the first job is started.

QUEUE is the program used to interface to the CMU mini-batch system. It is invoked by the QUEUE command or the SUBMIT command. The QUEUE command without arguments is the same as "R QUEUE". The SUBMIT command without arguments is the same as "QUEUE SUBMIT". The available queue-commands are HELP, SUBMIT, CANCEL, DESCHEDULE, REMIND, INSPECT, and PRESENT. The SUBMIT command submits a request for a job to be run. The CANCEL command cancels a previous request. If the job has started running, it is killed. The DESCHEDULE command kills the job if it has started running, but it does not delete the job request. The REMIND command, when fully implemented, will submit a request for SENDs to be issued to a subset of users at some future time. The INSPECT command displays the status of the BATCH queue. The PRESENT command describes the status of the presently running job(s). If no parameters are given to SUBMIT, REMIND or CANCEL then you will be prompted for them. It is possible to avoid the prompting and specify all parameters on a single command line. Follow the monitor command QUEUE by a queue-command and its parameters, or the monitor command SUBMIT by the parameters for the queue-command SUBMIT. The monitor will accept some queue-commands from terminals that are not logged in. Queue-commands have the format:

```
<qcommand> <filename> <arguments>
```

where <qcommand> is SUBMIT, REMIND, INSPECT, CANCEL, DESCHEDULE, or PRESENT. SUBMIT, CANCEL and DESCHEDULE require a file name which is the name of the control file. REMIND requires a target for the message. INSPECT and PRESENT do not take any arguments. The <arguments> are specified as <keyword><parameter> where the <keyword>s are

At time	(hh:mm) when to start execution
@ time	(same as AT)
On date	the day to start execution (no internal blanks)
To logfile	Defaults to <filename>.LOG
Running time	(h:mm:ss) maximum execution time
Stream	{Fast|Slow|1|2} Restrict this BATCH job to run only in the indicated stream.
PERmitting n	Allows n errors
For ppn	Defaults to user's ppn
Mayrestart	BATCH will restart job if it fails to complete due to a system crash.
ERrors	This word is ignored (see example below)
PAssword	If not supplied will be asked for
EVery	(hh:mm:ss or <n>DAYS or <n>WEEKS or MONTH) how often to repeat a job or a remind
UNtil	day to stop repeats of reminds
UPto	time to stop repeats of reminds

Unique abbreviations are accepted.

The function of ASKOPR is to get disk packs and DECtapes mounted when running under the batch monitor. If the operator can satisfy the user's request then the program will exit normally and the user can assume that the device needed is available after the running of ASKOPR. If ASKOPR is notified by the operator that the request cannot be met, then it (ASKOPR) will issue a RUNUUO to LOGOUT. Therefore, the user, if is not over quota, can be assured that the batch monitor will ignore any commands in the .CTL file after the running of an unsatisfied ASKOPR request. Now the user may mount the disk pack or read and write from the DECtape.

Examples of what could be in the .CTL file:

```
1.      R ASKOPR                                ; to the monitor
        MOUNT DISK SCRT                         ; to ASKOPR
        MOUNT SCRT                              ; to the monitor
        TYPE SCRT:FILE.EXT
2.      R ASKOPR
        MOUNT TAPE ZG03-12 AS DT12 ENABLED MESSAGE "TAPE IS IN C. ROOM"
        DIRE DT12:
3.      R ASKOPR
        REMOVE TAPE ZG03-12 FROM DT12
        DEASSI DT12
```

Note: Disk packs must be MOUNTed by the user after running ASKOPR, but tape drives will be assigned and given logical names as specified in the MOUNT TAPE command. Since, the user cannot find out the physical device (DTA0, DTA1, ..., DTA5) that was assigned, all references to the tape drive will have to be via the logical name. It is polite for the user to have his DECtapes dismounted when he is through with them (and safer if they are write enabled). When he is done with a DECtape drive the user should DEASSIGN that drive by its logical name.

Unique abbreviations are accepted for all commands. Arguments may contain no embedded blanks or tabs. The single exception to this is that the argument to MESSAGE may contain embedded blanks since it is enclosed in "'s. Its function is to print additional messages to the operator along with mount or dismount request.

4.10 The ARPANET

4.10.1 Overview

The ARPANET is an operational, computerized, packet switching DOD digital network which provides a capability for terminals or geographically separated computers, called Hosts, to communicate with each other. The Host computers typically differ from one another in type, speed, word length, and operating system. Each terminal or Host computer is connected into the network through a local small computer called an IMP or TIP. The complete network is formed by interconnecting the IMP's through wideband communication lines (50KB) supplied by common carriers.

Each IMP and TIP is programmed to receive and forward messages to the neighboring IMP's or TIP's in the network. During a typical Host to Host operation, a Host passes a message to its IMP; the message is passed from IMP to IMP through the network until it finally arrives at the destination IMP, which passes it along to the destination Host. A terminal (teletypewriter or CRT) accesses the network through a Terminal Interface Processor (TIP) or through a local host connected to the local IMP, and sends a message to a foreign Host or another terminal. The terminal message passes through the network in the same manner as a Host to Host message. The elements of the network are the Interface Message Processor (IMP), Terminal Interface Processors (TIP), interswitch trunks, access lines, host computers and terminals.

The ARPANET is intended to be used solely for the conduct of, or in support of, official U.S. Government business.

The use of the ARPANET must not be in violation of information privacy laws and is not intended to compete with existing commercial services.

4.10.2 IMPCOM (Telnet)

The primary function of IMPCOM is to provide terminal access to network hosts, either local (CMU) or remote. The TELNET command to IMPCOM connects your terminal to a specified host. For a new connection, the only required specification is the remote host name. If the remote host is willing, TELNET will then (1) follow the network Initial Connection Protocol to open a duplex connection to that host, (2) logically disconnect your terminal from your local job, and (3) connect it as a terminal of the remote host through an IMP device whose logical name is TELNET. You may then log on, etc., just as if you were at the remote site. Typing the local escape character (initially ↑← or control-shift-O) will cause your terminal to be disconnected from the remote host and reconnected to your local job. This does not, however, break your network connection, and you can reconnect to the remote host by again typing

TELNET host-name

An extensive IMP help facility may be invoked at monitor level by typing IMP HELP <arg>, where <arg> is an optional argument whose actual parameters may best be discovered by typing IMP HELP with no parameters. The same help facility is also available within the imp server. Ie, merely type HELP <arg> at any time while in * mode running IMP. The entire help file may be listed as SYS: IMPCOM.HLP

4.10.3 FTP (File transfer)

File transfer is done by two programs. The first is this program (FTP) which is run by the user and acts as an interface between the user and the network. The second is the server end (the server) which runs at the foreign site and acts as the interface between the foreign file system and FTP. These two programs communicate through a software entity called a 'socket'. FTP sends ASCII messages to the server, which replys with ASCII messages. FTP will print only the more important messages on the user's terminal. Once a proper connection is established with the server site, files may be retrieved from or stored onto it.

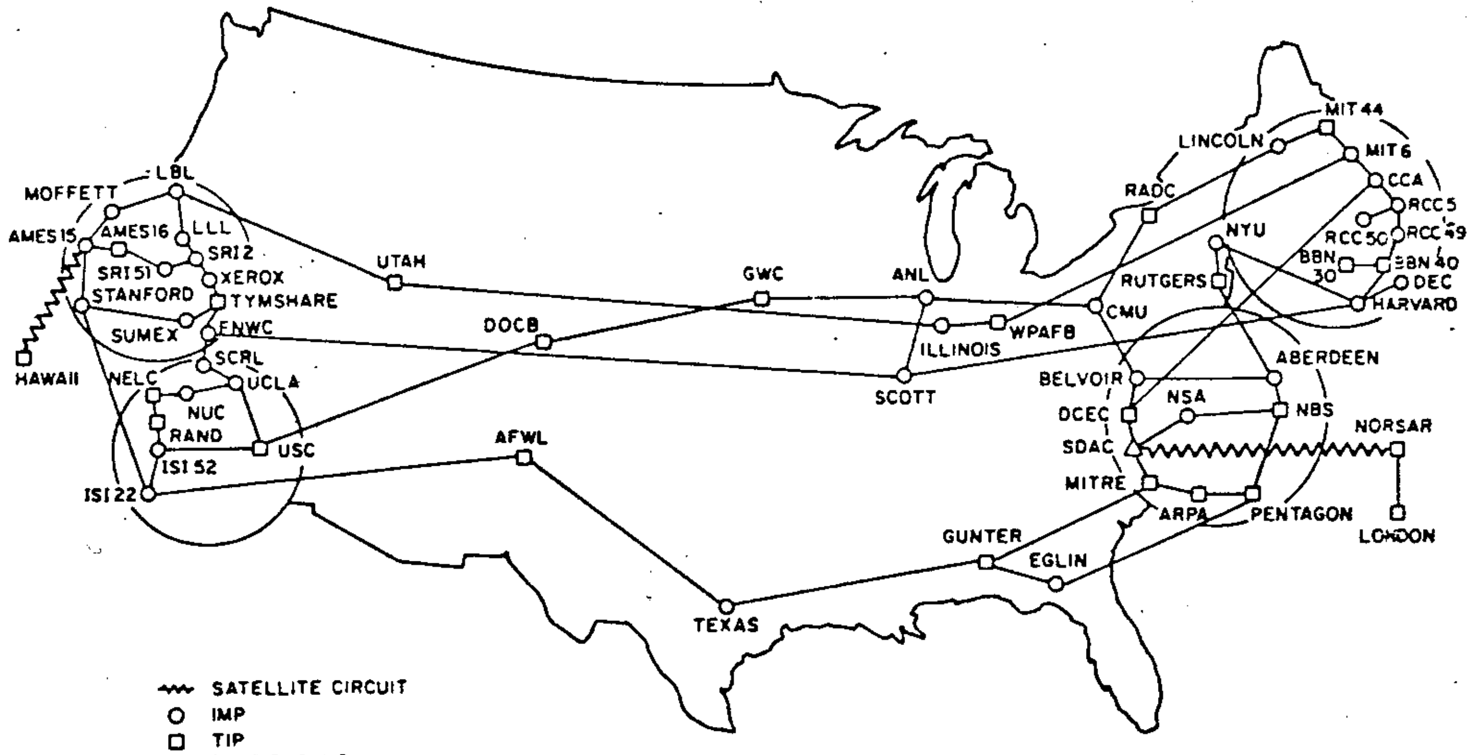
There are three levels of FTP commands which are relevant to most users. These are Network Control, Server Control, and File Transfer Control. They are usually invoked in that order.

```
Network Control:
        HOST    <name> | <num>  ; IMPCOM nick-name or decimal num
        QUIT                    ; in ! mode, quit host;
                                ; in * mode, quit FTP program
Server Control:
        USER    <username>      ; hopefully obvious in meaning
        PASS    <yourpsw>       ; for security's sake
        ACCT    <num>           ; for TENEX sites
        MAIL    <username>      ; terminate mail with ↑Z
        CPATH   <username>      ; change path to <username>'s UFD

File Transfer Commands:
        STOR    <filename>      ; push <filename> into remote host
        RETR    <filename>      ; retrieve <filename> from host
        LIST    LPT:;<ufd>      ; list directory of <ufd> on LPT:
        AUTO    <file.ext>      ; process command file <file.ext>
                                ; Note: ext must not be blank
```

See DOC:FTP.DOC or use help facilities within FTP for more information.

ARPANET GEOGRAPHIC MAP, JANUARY 1977

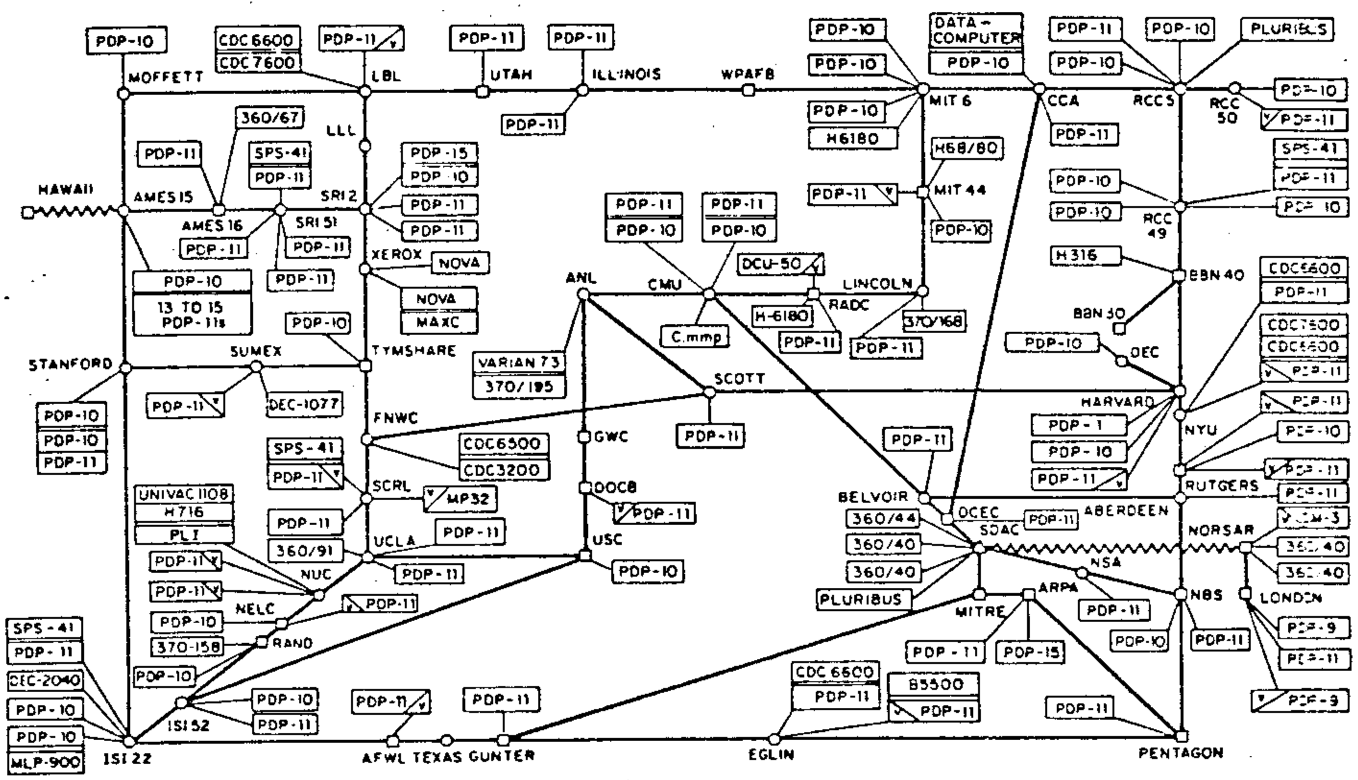


〰 SATELLITE CIRCUIT
○ IMP
□ TIP
△ PLURIBUS IMP

(NOTE: THIS MAP DOES NOT SHOW ARPA'S EXPERIMENTAL SATELLITE CONNECTIONS)

NAMES SHOWN ARE IMP NAMES, NOT (NECESSARILY) HOST NAMES

ARPANET LOGICAL MAP, JANUARY 1977



○ IMP △ PLURIBUS IMP
□ TIP 〰 SATELLITE CIRCUIT

(PLEASE NOTE THAT WHILE THIS MAP SHOWS THE HOST POPULATION OF THE NETWORK ACCORDING TO THE BEST INFORMATION OBTAINABLE, NO CLAIM CAN BE MADE FOR ITS ACCURACY)

NAMES SHOWN ARE IMP NAMES, NOT (NECESSARILY) HOST NAMES

5. Special purpose research systems

Some of the research efforts at CMU require stand-alone computers. Such projects usually have hardware constraints or configurations that are unsuitable for the general purpose time sharing systems. These independent systems often become vehicles for operating system and architechture experimentation by students and staff. Three of these machines are described below by people currently working on them.

Cm*: A Multi-Micro Processor

Don Scelza, Aug. 2, 1978

Cm* is a multi-processor under development at C-MU. The machine is still very much in the developmental stages hence access to Cm* is limited to users doing research in distributed operating system design and distributed hardware design.

When connecting to Cm* from the Front End you are communicating with the Cm* Host System. This is an external processor that allows communication between the Cm* machine and the outside world.

At the present time the Cm* machine has the following configuration:

Processors:	11 LSI-11's in the Cm* machine 2 LSI-11's acting as Hooks into the Kmaps 3 C-MU built mapping processors (Kmap's) 1 PDP-11/10 Host processor
Memory:	324K words
Links:	2 9600 baud links to the CMU10B 4 9600 baud links to the CMU10D 1 1200 baud link to the CMU Front End via the Host
DECtapes:	4 which are accessed through the Host
Terminals:	4 hardwired to the Host Any number of virtual Front End connections

UNIX: AI Speech and AI Vision

Steven Rubin, July 28, 1977

The Computer Science Department has two machines that run the UNIX timesharing system for a small community of Artificial Intelligence researchers. The speech machine, which has the most accurate DAC/ADC devices at C-MU, is used primarily for speech understanding systems and music generation. The vision machine has a number of high-bandwidth devices which make it suitable for image understanding work.

The true names of the machines are "SUS" and "IUS" (Speech Understanding System and Image Understanding System) but their Front End names are "AI Speech" and "AI Vision", respectively.

The machines are configured as follows:

	AI Speech	AI Vision
Processors	1 PDP-11/40E	2 PDP-11/40Es
Writable Microstore	Yes	Yes
Memory	128K words	512K words
Memory control	DEC Memory Management	C.mmp-style reloc on 4x4 switch
Disks	2 RP02s, 1 RP06	2 RP06s
Magtape	9-track, 800 BPI	9-track, 800 BPI
DECtapes	2	2
Digital-to-Analog	Yes	No
Analog-to-Digital	Yes	No
On Front End	Yes	Yes
On Arpa Net	No	No
Printer	None	Gould Printer/Plotter
Displays	Graphics	Color Video
Links	CMU10B	CMU10D, CMU Front End
Hard-wired terminals	2	2

Since heavy use of the machines tends to grind them to a halt, UNIX accounts are restricted to those people who can justify their use of the machines.

ALGOL

T. Teitelbaum, L. Snyder, J. Dills
(Revised Jan. 1973)

Algol 60 is an algebraic programming language developed by an international committee in 1960. Algol was designed at a time when many computer installations had their own ad hoc algebraic programming languages. Algol was intended to be a machine independent standard for the communication (and execution) of algorithms. Most of the arbitrary restrictions found in languages such as FORTRAN were eliminated. Algol was the first language for which a complete and precise syntactic and semantic definition was attempted. The terminology used in this definition (in the Algol Report) has come into wide use in computer science. Algol is characterized by dynamic array allocation, recursive procedures, block structure, and a generalized parameter passing mechanism.

REFERENCES

Manual

[1]	Digital Equipment Corp.	DECSYSTEM 10 MATHEMATICAL LANGUAGES HANDBOOK

Definition

[2]	Naur, P. (ed.)	Revised Report on the Algorithmic Language ALGOL 60. Comm.ACM 6 (Jan 63).
[3]	Knuth, D. E.	The remaining trouble spots in ALGOL 60. Comm.ACM 10 (Oct 67).
[4]	Abrahams, P. W.	A final solution to the dangling else of ALGOL 60 and related languages. Comm.ACM 9 (Sept 66).
[5]	Knuth, D. E., Merner, J. N.	ALGOL 60 Confidential. CACM, Vol. 4, 1961.

Philosophy

[6]	Perlis, A. J.	The synthesis of algorithmic systems. J.ACM 14 (Jan 1967).

History-Bibliography

[7]	Bemer, R. W.	A politico-social history of ALGOL. Annual Review In Automatic Programming, 5 Pergamon Press, 1969.
[8]	Sammet, Jean	Programming Languages: History and Fundamentals, Prentice-Hall, 1969.

Introductory

[9]	Bottenbruch, H.	Structure and use of ALGOL 60. J.ACM 9 (Apr 62).
[10]	Higman, B.	What everybody should know about ALGOL. Computer Journal 6 (1963) p. 50.
[11]	Ekman, T. and Froberg, C.	Introduction to ALGOL programming. Oxford University Press, (1967).
[12]	Dijkstra, E. W.	A Primer of ALGOL 60 Programming, Academic Press, London, 1962.

Implementation

[13]	Evans, A.	An ALGOL 60 Compiler. Annual Review in Automatic Programming, 4. Pergamon Press (1964).
[14]	Randell, B. and Russell, L. J.	ALGOL 60 Implementation. Academic Press, (1964), 418 pp.
[15]	Dijkstra, E. W.	"Making a Translator for ALGOL 60," Annual Review of Automatic Programming, Vol. 3., MacMillan, 1963, pp. 347-356.

Extensions

[16]	Wirth, N.	A Generalization of ALGOL. Comm.ACM 6 (Sept 63).
[17]	Perlis, A. J. and Iturriaga, R.	An extension to ALGOL for manipulating formulae. Comm.ACM 7 (Feb 64).
[18]	Wirth, N. and Weber, H.	EULER: A generalization of ALGOL and its formal definition. Comm.ACM 9 (Jan, Feb 66).
[19]	Wirth, N. and Hoare, C. A. R.	A contribution to the development of ALGOL. Comm.ACM 9 (June 66).
[20]	Dahl, O. J. and Nygaard, K.	SIMULA -.An ALGOL-based simulation language. Comm.ACM 9 (Sept 66).
[21]	Hoare, C. A. R.	Record Handling, in F. Genays (Ed.) Programming Languages, Academic Press, 1968, pp. 291-347.

SAMPLE PROBLEMS

1. Continued Fractions

Let $Q_1 = \cfrac{1}{1+1}$, $Q_2 = \cfrac{1}{1+\cfrac{1}{1+1}}$, $Q_3 = \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+1}}}$, etc.

As $i \to \infty$, $Q_i \to Q = 0.61803...$

Write an ALGOL 60 function procedure Phi (n) that will return the value Q_n. For example, Phi (2) = 0.6666. Write two versions of Phi, one recursive and the other iterative.

2. Palindromes

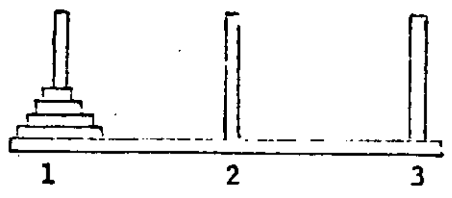
A palindrome is a vector V of values such that V = XY where X = reversal of Y. E.g., 110011.

Write a Boolean function that determines if a vector is a palindrome.

Write another which determines if a vector consists of a list of palindromes; e.g., 110110.

3. Tower of Hanoi

Write an ALGOL program to print the solution sequence to the towers of Hanoi puzzle. Given,



Move the stack of disks on pin 1 to pin 2 (possibly using pin 3 as intermediate storage) so that (1) the disks finally end up in the same order as they started (as shown); (2) at no time is a large disk on top of a smaller disk; and (3) only one disk at a time is moved. Your program should allow an arbitrary number of disks.

4. Partitions

Write an ALGOL procedure PART(X) which prints the partitions of the integer X. A partition is defined as a sequence of positive integers which sum to X. If that's too easy, find the unique partitions of X.

5. Pascal's Triangle

Recall that Pascal's triangle begins:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
. . . . .
```

Write an ALGOL procedure, PASCAL(N), which prints the Nth row of Pascal's triangle. It should be possible to compute the result without a factorial routine and with only a single vector for a data structure.

6. Pattern of Primes

Write a program which fills an N x N array A with the integers 1 through N^2 arranged in a spiral.

E.g., when N = 3, then A =

```
7 8 9
6 1 2
5 4 3
```

The pattern of primes in this arrangement (for large N) has been of some interest (to some people). Try a printout where primes are '*' and non-primes blank.

Can you think of a more efficient storage arrangement for the pattern of primes when N is large?

7. How well do you understand call-by-name and call-by-value?

```
BEGIN REAL A,B;
  REAL PROCEDURE INCV(X);VALUE X;REAL X;
        BEGIN X←X+1; INCV←X END;
  REAL PROCEDURE INCN(X);REAL X;
        BEGIN X←X+1; INCN←X END;
  REAL PROCEDURE ADDV(Y);VALUE Y;REAL Y;
        ADDV←Y+Y;
  REAL PROCEDURE ADDN(Y);REAL Y;
        ADDN←Y+Y;

  A←1; B←ADDV(INCV(A));
  COMMENT A IS NOW ------, B IS NOW -----;

  A←1; B←ADDV(INCN(A));
  COMMENT A IS NOW ------, B IS NOW -----;

  A←1; B←ADDN(INCV(A));
  COMMENT A IS NOW ------, B IS NOW -----;

  A←1; B←ADDN(INCN(A));
  COMMENT A IS NOW ------, B IS NOW -----;
END;
```

8. Exchange

Write a procedure EXCH(A,B) that exchanges A and B. This is not as easy as it seems. Consider the problems exchanging I and A[I].

Answers for odd number problems follow the Algol Script.

Algol Script

In this script I use the text editor SOS to create and edit files. The SOS commands which I use here are I,D,R,P,E which are insert,delete,replace,print, and end, respectively. To find out more about these commands, see the SOS part of this document.

```
.CREATE FIB.ALG
00100   BEGIN
00200   INTEGER PROCEDURE FIBONACCI(N);VALUE N;INTEGER N;
00300   BEGIN
00400   IF N<=1
00500     THEN FIBONACCI:=1
00600     ELSE FIBONACCI:=FIBONACCI(N-1)+FIBONACCI(N-2);
00700   END;
00800   READ(K);                 this program calculates Fibonacci
00900   J:=FIBONACCI(K);         numbers by using a recursive procedure.
01000   PRINT(J,6);              It should be noted that this isn't
01100   END                      a very efficient method.
01200   $       this is an altmode
*E

EXIT

.R ALGOL                  here the algol compiler is called.
*FIB,TTY:←FIB             this line causes the object file to be
                          named FIB.REL, the listing to be placed
                          on the teletype, and the source file to
                          be named FIB.ALG.

DECSYSTEM 10 ALGOL-60, V. 2B(146):
17-JUN-72        03:00:00


00100   BEGIN
00200   INTEGER PROCEDURE FIBONACCI(N);VALUE N;INTEGER N;
00300   BEGIN
00400   IF N<=1
00500     THEN FIBONACCI:=1
00600     ELSE FIBONACCI:=FIBONACCI(N-1)+FIBONACCI(N-2);
00700   END;
00800   READ(K);
********     ↑
800  UNDECLARED IDENTIFIER              the error messages would
00900   J:=FIBONACCI(K);                appear on the TTY even
******** ↑                              if the file wasn't listed
900  UNDECLARED IDENTIFIER              there.
01000   PRINT(J,6);
01100   END


?2 ERRORS

*↑C                          control C out of the compiler
```

```
.EDIT FIB.ALG
*I150
00150    INTEGER J,K;
*E

EXIT

.EX FIB                       EX is a CCL command which causes
ALGOL: FIB                    compiling,loading and execution.
LOADING

LOADER 1K CORE
EXECUTION
10                  I enter a 10
     89             program returns 89, the 10th Fibonacci number.

END OF EXECUTION - 2K CORE

EXECUTION TIME: 0.17 SECS.

ELAPSED TIME:   11.88 SECS.

                              The following procedure is a pseudo random
                              number generator.  More information about
                              this type of algorithm can be found in
                              The Art of Computer Programing, Vol 2 by
                              Don Knuth.
.CREATE RAND.ALG
00100    INTEGER PROCEDURE RAND(LESS);VALUE LESS;INTEGER LESS;
00200    COMMENT THIS PROCEDURE RETURNS AN INTEGER BETWEEN 0 AND LESS-1;
00300    BEGIN OWN INTEGER SEED,MUL,MOD;
00400    IF SEED=0                 In this algol compiler, own variables
00500      THEN  BEGIN             are initialized to 0.
00600            COMMENT SEED,MUL&MOD MUST BE LESS THAN 185364
00700            TO PREVENT OVERFLOW.  THE FOLLOWING NUMBERS
00800            ARE 7↑6,5↑7,2↑17 RESPECTIVELY;
00900            SEED←117649;
01000            MUL←78125;
01100            MOD←131072;
01200            END;
01300    SEED←(SEED*MUL) REM MOD;          this line is the generator
01400    RAND←(LESS*SEED) DIV MOD;
01500    END
01600    $
*E

EXIT

                              This program will read numbers and print a
                              random number less than the number just read
                              until a zero is read. The random numbers are
                              generated by the above procedure
                              which is called externally.
.CREATE TESTR.ALG
00100    BEGIN INTEGER I,R;
00200    EXTERNAL INTEGER PROCEDURE RAND;
00300    READ(I);
```

```
00400   WHILE I#0 DO
00500           BEGIN
00600           R:=RAND(I);
00700           PRINT(R,3);
00800           NEWLINE;
00900           BREAKOUTPUT;
01000           READ(I);
01100           END;
01200   END
01300   $
*E

EXIT
```

NEWLINE causes a carriage return and a line feed to be placed in the output buffer. BREAKOUTPUT causes the output buffer to be dumped to the output device, in this case the TTY.

```
.EX TESTR,RAND
ALGOL: TESTR
ALGOL: RAND
LOADING

LOADER 1K CORE
EXECUTION
100
  26
100
   4
100
  93
100
   2
0

END OF EXECUTION - 2K CORE

EXECUTION TIME: 0.03 SECS.

ELAPSED TIME:   42.78 SECS.
```

The program is altered so that it will produce a number of random numbers, all from the same range.

```
.EDIT TESTR.ALG
R100
00100   BEGIN INTEGER L,I,J,R;
*I250
00250   WRITE("RANGE:");BREAKOUTPUT;
*I325,25
00325   WRITE("NUMBER:");BREAKOUTPUT;
00350   READ(L);
00375   $
*R400
00400   FOR J:=1 UNTIL L DO
00425   $
*D800
*D1000
*E

EXIT
```

here I delete NEWLINE, note the effect below.

```
.EX TESTR,RAND
ALGOL: TESTR
LOADING

LOADER 1K CORE
EXECUTION
RANGE:100                    the program prints RANGE: and NUMBER:
NUMBER:50                    I respond with 100 and 50.
  26   4  93   2  84  27  55  30   3  86  92  27  75  60  21  96  40  75
   7  90  89  12  66  13  62  96  64  17  50   8  15   6  94  75  87  32
  84  74  93  99  61  35   1  60  16  35  25  19  89   2
                             the program then prints 50 random numbers
END OF EXECUTION - 2K CORE             less than 100.

EXECUTION TIME: 0.35 SECS.

ELAPSED TIME:   32.10 SECS.

.EDIT TESTR.ALG
*I360,10
00360   OUTPUT(4,"DSK");                assigns the disk to channel 4
00370   SELECTOUTPUT(4);                selects channel 4 for output
00380   OPENFILE(4,"RAND.DAT");         opens a file named RAND.DAT
00390   $
*E

EXIT

.EX TESTR,RAND
ALGOL: TESTR
LOADING

LOADER 1K CORE
EXECUTION
RANGE:100
NUMBER:50

END OF EXECUTION - 2K CORE

EXECUTION TIME: 0.22 SECS.

ELAPSED TIME:   11.60 SECS.

.TYPE RAND.DAT
  26   4  93   2  84  27  55  30   3  86  92  27  75  60  21  96  40  75
   7  90  89  12  66  13  62  96  64  17  50   8  15   6  94  75  87  32
  84  74  93  99  61  35   1  60  16  35  25  19  89   2
.EDIT TESTR.ALG
*D250
*D325
*I225,25
00225   INPUT(3,"DSK");                 Here changes are made so that
00250   SELECTINPUT(3);                 the range and number can be read
00275   OPENFILE(3,"RANGE.NUM");        from a file named RANGE.NUM.
*E

EXIT
```

```
.CREATE RANGE.NUM
00100   100
00200   50
00300   $
*E

EXIT

.R PIP
*RANGE.NUM/N←RANGE.NUM
*↑C
.EX TESTR,RAND
ALGOL: TESTR
LOADING

LOADER 1K CORE
EXECUTION

FATAL RUNTIME ERROR AT ADDRESS 000167

MORE HEAP SPACE REQUIRED FOR I-O BUFFERS

?ACTION (H FOR HELP)? F


END OF EXECUTION - 2K CORE

EXECUTION TIME: 0.05 SECS.

ELAPSED TIME:   17.33 SECS.
```

Here I strip the line numbers off the data file, because algol can't handle line numbers on data files.

Heap size must be increased when 2 or more disk files are being used, because the default size is too small.

```
.R ALGOL
*TESTR,←TESTR/600D
*↑C
.EX TESTR,RAND
LOADING

LOADER 1K CORE
EXECUTION


END OF EXECUTION - 2K CORE

EXECUTION TIME: 0.13 SECS.

ELAPSED TIME:   3.13 SECS.

.TYPE RAND.DAT
  26   4  93   2  84  27  55  30   3  86  92  27  75  60  21  96  40  75
   7  90  89  12  66  13  62  96  64  17  50   8  15   6  94  75  87  32
  84  74  93  99  61  35   1  60  16  35  25  19  89   2
```

Here the heap size is increased to 600 blocks which is enough for 2 disk files.

```
.CREATE TEST.ALG
BEGIN INTEGER I,N;STRING STR;
STRING PROCEDURE TERM;
BEGIN STRING T;INTEGER S,I;
T:=NEWSTRING(50,7);
INSYMBOL(S);
WHILE S<=32 DO INSYMBOL(S);
T.[1]:=S;
I:=1;
WHILE S>32 AND I<50 DO
        BEGIN
        INSYMBOL(S);
        I:=I+1;
        T.[I]:=S;
        END;
IF S<=32 THEN I:=I-1;
TERM:=COPY(T,I);
DELETE(T);
END;
WRITE("ENTER:");BREAKOUTPUT;READ(N);
FOR I:=1 UNTIL N DO
        BEGIN
        STR:=TERM;
        WRITE(STR);
        END;
BREAKOUTPUT;
END;
$
*E

EXIT

.EX TEST
ALGOL: TEST
LOADING

LOADER 1K CORE
EXECUTION
ENTER:5
CARNEGIE -MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
CARNEGIE-MELLONUNIVERSITYCOMPUTERSCIENCE

END OF EXECUTION - 2K CORE

EXECUTION TIME: 0.18 SECS.

ELAPSED TIME:   1 MINS. 35.12 SECS.
```

When using READ to read strings, it is necessary to put quote marks around each string that is read. This program demonstrates a procedure that will read, without quotes, a string of characters up to the next break character.

```
.CREATE BAT.ALG
00100   BEGIN
00200   RECORD CLASS NODES[20](RECORD NEXT OF NODES;INTEGER AB,HITS;
00300           STRING NAME);
00400   RECORD NODE,TOP OF NODES;
00500   TAKE(NODE);
00600   READ(NODE@NAME);
00700   READ(NODE@AB);
00800   READ(NODE@HITS);
00900   WHILE NODE@NAME#"" DO
01000           BEGIN
01100           NODE@NEXT←TOP;
01200           TOP←NODE;
01300           TAKE(NODE);
01400           READ(NODE@NAME);
01500           READ(NODE@AB);
01600           READ(NODE@HITS);
01700           END;
01800   NODE←TOP;
01900   WHILE NOT NULL(NODE) DO
02000           BEGIN
02100           WRITE(NODE@NAME);
02200           PRINT((NODE@HITS/NODE@AB),3,3);
02300           NEWLINE;
02400           NODE←NODE@NEXT;
02500           END;
02600   END
02700   $
*E
```

This program demonstrates the use of record classes. They are an addition to algol here at CMU. They are useful for constructing complex data structures, such as the linked list of baseball players in this program.

```
EXIT

.EX BAT
ALGOL: BAT
LOADING

LOADER 1K CORE
EXECUTION
"COBB" 11429 4191
"RUTH" 8399 2873
"WILLIAMS" 7706 2654
"" 0 0
WILLIAMS 0.344
RUTH 0.342
COBB 0.367

END OF EXECUTION - 2K CORE

EXECUTION TIME: 0.17 SECS.

ELAPSED TIME:    45.27 SECS.

.
```

Further documentation on record classes is avaiable in the departmental office.

End of Algol Script.

Solutions to Sample Problems

1.
```
REAL PROCEDURE PHIR(N);VALUE N;INTEGER N;
     PHIR←IF N=0 THEN 1.0 ELSE 1.0/(1.0+PHIR(N-1));
REAL PROCEDURE PHII(N);VALUE N;INTEGER N;
     BEGIN REAL P;P←0.0;
     WHILE (N←N-1) > 0 DO P←1.0/(1.0+P);
     PHII←P;END;
```

3.
```
PROCEDURE HANOI(N,START,OTHER,FINISH);
  VALUE N,START,OTHER,FINISH;INTEGER N,START,OTHER,FINISH;
     BEGIN IF N=1 THEN BEGIN
          WRITE("MOVE DISC    1 FROM");PRINT(START,3);
          WRITE(" TO");PRINT(FINISH,3);NEWLINE;END
     ELSE BEGIN
          HANOI(N-1,START,FINISH,OTHER);
          WRITE("MOVE DISC");PRINT(N,3);WRITE(" FROM");
          PRINT(START,3);WRITE(" TO");PRINT(FINISH,3);NEWLINE;
          HANOI(N-1,OTHER,START,FINISH);END;
     BREAKOUTPUT;END;
```

5.
```
PROCEDURE PASCAL(N);VALUE N;INTEGER N;
     BEGIN INTEGER ARRAY P[1:N];INTEGER I,J;
     P[1]:=1;
     FOR I:=2 UNTIL N DO
          BEGIN P[I]:=0;
          FOR J:=I STEP -1 UNTIL 2 DO P[J]:=P[J]+P[J-1];
          END;
     FOR I:=1 UNTIL N DO PRINT(P[I],4);
     END;
```

7.
```
1.0  4.0
2.0  4.0
1.0  4.0
3.0  5.0
```

A reprint from **computers and automation** *November, 1969* *Vol. 18, No. 12*

APL: A PERSPICUOUS LANGUAGE

Garth H. Foster
Department of Electrical Engineering
Syracuse University
Syracuse, N.Y. 13210

"In APL, a great many highly useful functions which are required in computing have been defined and given a notation consisting of a single character."

The news and promotion copy now beginning to appear in many computer-related publications proclaiming APL (*A* *P*rogramming *L*anguage) to be everything from a successor to PL/I (Programming Language One) to the most powerful interactive terminal system available, has no doubt been widely noticed. Such copy has led many to wonder what APL is, and after seeing its notation, many wonder about its clarity.

This article is not intended to a tutorial on APL, for that would take more space than is warranted here. However, let us discuss some of the aspects of APL which have excited the academic communities at a number of colleges and universities and at least one high school system, and which have triggered a number of implementation efforts in Canada, France, and the United States. The interested reader may then investigate further the many features of APL which cannot all be covered here. To assist in this direction, a rather complete bibliography of APL source material is appended to this article.

Definition

The initials APL[1] derive from the title of the book "A Programming Language" by K.E. Iverson, published by John Wiley and Sons in 1962; and it was that publication which served as the primary vehicle for the publication of the initial definition of APL. Subsequent development of the language by Iverson has been done in collaboration with A.D. Falkoff at IBM's Thomas J. Watson Research Center, Yorktown Heights, New York.

The present form of APL is the APL\360 Terminal System, the implementation of APL on the system 360. Although there are implementations for the IBM 1130 and 1500 computers, when we speak of APL we shall mean APL\360.

The terminal system was designed by Falkoff and Iverson with additional collaboration from L.M. Breed, who, with R.D. Moore (I.P. Sharp Associates, Toronto) developed the implementation. Programming was by Breed, Moore, and R.H. Lathwell, with continuing contributions by L.J. Woodrum (IBM, Poughkeepsie), and C.H. Brenner, H.A. Driscoll, and S.E. Krueger (SRA, Chicago). Experience had been gained from an earlier version which was created for the IBM 7090 by Breed and P.S. Abrams (Stanford U., Stanford, California).

A computer language which is classified as algebraic is generally, but not exclusively, used to program problems requiring reasonably large amounts of arithmetic. Generally such languages have available, as formalized arithmetic operators with a notation, the operations of addition, subtraction, multiplication, division, and exponentiation; and there the list ends. To achieve other arithmetic operations either calls to pre-written subroutines must be made or the user must supply his own.

This is not true of APL; a great many highly useful functions which are required in computing have been defined and given a single character notation (some of these require 3 keystrokes, striking a key, backspacing and then striking another key; but usually only a single keystroke is required.)

The APL Keyboard

Figure 1 shows the APL keyboard. The letters and numbers all appear in their usual places on a typewriter, except that the capital letters are in the lower case positions (the lower case letters do not appear). The up-shift positions on the keyboard are occupied by symbols used to represent the powerful set of APL operators.

[1]APL should not be confused with "ABL — A Language for Associative Data Handling in PL/I," by George G. Dodd, General Motors Research, 1966 Fall Joint Computer Conference.

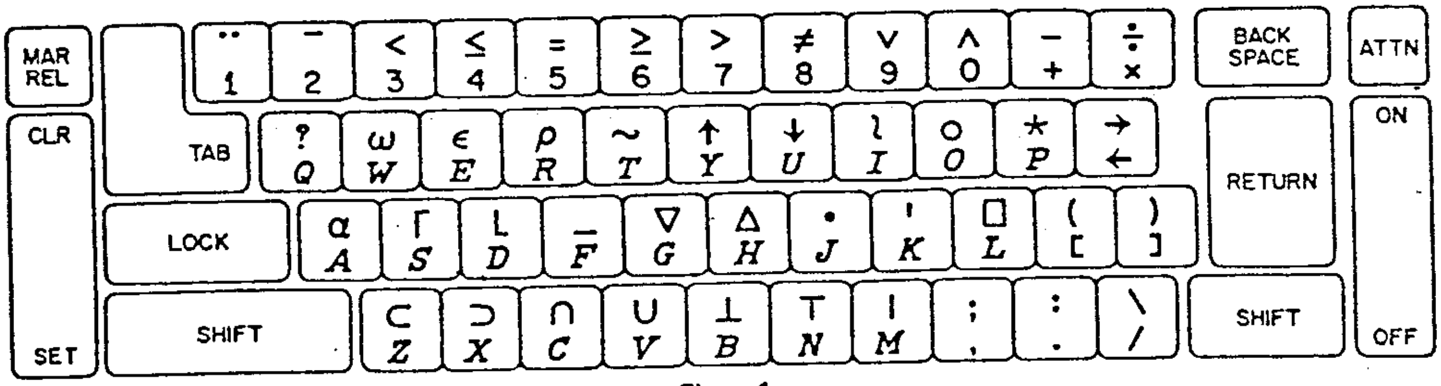


Figure 1

Besides +, -, x, ÷, (the familiar symbols for addition, subtraction, multiplication, and division located on the two right-most keys on the top row) and the symbol * assigned to represent exponentiation (the star over the P as in raising to a power), there are distinct single character notations for the operations of: negation; signum; reciprocal; logarithms (to both natural and arbitrary base); combinations and factorials; base e raised to a power; the residue of a number modulo any divisor. There are characters which represent taking: PI times a number; sines; cosines; tangents; hyperbolic sines, cosines, and tangents; and the inverse functions for the six preceding functions. Available too are: floor (truncating a number to the largest integer less than or equal to the number); ceiling (rounding up to the smallest integer greater than or equal to the number); and maximum or minimum of a pair of numbers.

APL also provides the relations which test whether two numbers are: less than; less than or equal to; greater than or equal to; greater than; equal; or not equal. The last two relations are also applicable to characters. These relations check to see, for example, if a relation is true and produce 1 (representing TRUE) or 0 (FALSE); these binary quantities may be operated upon by the logical functions of: OR; AND; NOT; NOR; and NAND. All these are also available as standard functions in APL, and are designated by a single character graphic. These operations are all summarized in Figure 2.

Monadic form ƒB		ƒ	Dyadic form AƒB	
Definition or example	Name		Name	Definition or example
+B ←→ 0+B	Plus	+	Plus	2+3.2 ←→ 5.2
-B ←→ 0-B	Negative	-	Minus	2-3.2 ←→ ¯1.2
×B ←→ (B>0)-(B<0)	Signum	×	Times	2×3.2 ←→ 6.4
÷B ←→ 1÷B	Reciprocal	÷	Divide	2÷3.2 ←→ 0.625
B ⌈B ⌊B 3.14 4 3 ¯3.14 ¯3 ¯4	Ceiling	⌈	Maximum	3⌈7 ←→ 7
	Floor	⌊	Minimum	3⌊7 ←→ 3
*B ←→ (2.71828..)*B	Exponential	*	Power	2*3 ←→ 8
⍟*B ←→ B ←→ *⍟B	Natural logarithm	⍟	Logarithm	A⍟B ←→ Log B base A A⍟B ←→ (⍟B)÷⍟A

|¯3.14 ←→ 3.14	Magnitude	|	Residue	Case A|B
				A≠0 B-(|A)×⌊B÷|A
				A=0,B≥0 B
				A=0,B<0 Domain error

!0 ←→ 1 !B ←→ B×!B-1 or !B ←→ Gamma(B+1)	Factorial	!	Binomial coefficient	A!B ←→ (!B)÷(!A)×!B-A 2!5 ←→ 10 3!5 ←→ 10
?B ←→ Random choice from ⍳B	Roll	?	Deal	A Mixed Function
○B ←→ B×3.14159...	Pi times	○	Circular	See Table at left
~1 ←→ 0 ~0 ←→ 1	Not	~		
		∧	And	
		∨	Or	
		⍲	Nand	
		⍱	Nor	
		<	Less	Relations Result is 1 if the relation holds, 0 if it does not: 3≤7 ←→ 1 7≤3 ←→ 0
		≤	Not greater	
		=	Equal	
		≥	Not less	
		>	Greater	
		≠	Not Equal	

A	B	A∧B	A∨B	A⍲B	A⍱B
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	1	0
1	1	1	1	0	0

(-A)○B	A	A○B
(1-B*2)*.5	0	(1-B*2)*.5
Arcsin B	1	Sine B
Arccos B	2	Cosine B
Arctan B	3	Tangent B
(¯1+B*2)*.5	4	(1+B*2)*.5
Arcsinh B	5	Sinh B
Arccosh B	6	Cosh B
Arctanh B	7	Tanh B

Table of Dyadic ○ Functions

Figure 2

Order of Operations

Of course when such a host of generalized and powerful operations are at the disposal of the programmer, there is immediate concern as to the order or precedence of operations in an arithmetic expression written without parentheses.

Traditionally in algebraic languages, exponentiations were performed before multiplications and divisions, and these were done before additions and subtractions. One of the reasons for this choice (of hierarchy of operations) was that normal conventions in algebraic notation provided that the expression

$$5.6y^3 + 8y^2 + 2.84y + 9.06$$

could be written as

$$5.6 * y * * 3 + 8 * y * * 2 + 2.84 * y + 9.06$$

without the use of parentheses.

If one wanted to make the compiler work more efficiently when programming in the higher order language, then parens (parentheses) *were* used and the polynomial was "nested", so that in the above example one coded:

$$((5.6 * y + 8) * y + 2.84) * y + 9.06$$

That is to say, one discarded the built-in precedence order.

Clearly, in APL having all the functions shown in Figure 2, the establishment of any hierarchy of operators would be arbitrary and open to question at best; and more than likely it would border on the impossible to justify the hierarchy in any reasonable way.

Thus in APL there is only *one* rule for evaluating all unparenthesized expressions (or within a pair of parens), and that rule is.

> Every operator takes as its right-hand argument the value of everything to the right of it (up to the closing parenthesis).

Now such a rule may seem strange and unfamiliar to someone who is now programming, but it has advantages:

(1) Uniformity—it is applied in the same way for all standard or primitive functions provided by the APL system as well as all functions (programs) written in APL by the user;

(2) Utility—this approach, for example, allows the nested polynomial to be written without parentheses as:[2]

$$9.06 + Y \times 2.84 + Y \times 8 + Y \times 5.6$$

It is also possible to write continued fractions without parentheses and the rule given provides other interesting and useful results as a by product.

Sum Reduction

Another area in which looping (of computer instructions) is explicitly required in most programming languages but not in APL is that of summing the components of a vector, which we will call for the sake of example, X. The usual approach is to initialize the sum to zero and then use a running index variable of a DO or FOR loop, and then take the summation by an expression like

$$SUM = SUM + Z(I).$$

In APL we use what is called *sum reduction.* This is the name for *conceptually* taking the vector X, inserting plus signs between each of its components, and then evaluating the resulting expression; its notation is simply +/X. If we had wanted to take the product of the elements of a vector Q, then in APL we write x/Q and this provides the *times reduction.*

[2]There are even more powerful ways to evaluate a polynomial expression in APL, but the availability of such methods does not reduce the effectiveness of the right to left rule just described.

The Value of Powerful Operators

Thus the first area in which APL provides clarity in programming is by providing a large set of powerful functions. Now one may ask whether writing A⌈B in APL is only marginally more compact than say writing MAX(A,B). However, in APL we are allowed to use A!B to denote the combinations of taking B things A at a time. Such an operation in languages other than APL generally require the user to write his own program, perhaps calling upon routines to provide the factorials and if they in turn are not available, writing that routine also. The claim is that the presence of the APL operator ! in a program provides much more clarity than the presence of the equivalent routine in another programming language.

Of course one may argue that factorials and combinations are not needed all that much anyway. In many cases such a point of view may be correct; however, the fact still remains that the need for, say, the FORTRAN Library of subroutines indicates a need for arithmetic computations which are more complex than the operations included in the language as primitives. What APL has done therefore is to move in the direction of a library increasing the sophistication of the language, and at the same time simplifying the notation for using a much more powerful set of operators.

Extending the Scope of Functions

The next step forward which APL has taken is to extend the scope of those functions shown in Figure 2, in the following way. In most languages extant today, if one writes A + B, then one commands the computer to add the number A to the number B. In APL the command still produces the addition of the single numbers, called scalars, if that is the nature of the variables A and B. If on the other hand, A and B are each names for a collection or string of numbers, called a vector, then the addition takes place on an element by element basis, with the first element of A being added to the first element of B, the second to the second, and so forth. The requirement is that either A or B may be a scalar while the other is a vector, but if they are both vectors, then they must have the same number of elements, that is, they must be of the same size.

If A and B are matrices of the same size (having the same number of rows and columns), then A + B in APL adds, on an element by element basis, matrix A to matrix B. To perform equivalent operations in most computer languages requires a DO or a FOR loop when adding vectors, or nested loops when adding matrices.

Two comments are relevant here. First, the explicit loops embodied in the DO or FOR loops are required by the language, but they are ancillary to communicating the process to be performed, say adding two matrices. Second, the utility of providing an extension of this nature, where the system assumes additional responsibility, is borne out, for example, in the MAT commands of BASIC. APL extends such ideas and applies them uniformly to all data structures treated in the language. In fact, from the programmer's point of view, one does not care in what sequence the operations in the loops implied in such an APL command take place. They could just as well be done all in parallel; the fact that the computer does not process the matrix elements in parallel does not matter. The extension of scope of the notation allows the algorithm to be thought of as acting on the data in parallel. Thinking about the computing process in this way gives new insight into the way the programs manipulate or transform the data.

Allocating Space for Arrays

The philosophy is that the system should perform the tasks which are required by the computer but not essential to the algorithm. A useful extension is to have the computer assume the burden of allocation of space for arrays on a dynamic basis. This is done in the APL terminal system; for example, if one creates the vector X having components 2, 5, and 10, then X ← 2 5 10 is the *specification* or *assignment* of those constants to be the value of the variable X. No dimensioning is required. Later if we wish to respecify

Name	Sign[1]	Definition or example[2]
Size	ρA	ρP ↔ 4 ρE ↔ 3 4 ρ5 ↔ ⍳0
Reshape	VρA	Reshape A to dimension V 3 4ρ⍳12 ↔ E 12ρE ↔ ⍳12 0ρE ↔ ⍳0
Ravel	,A	,A ↔ (×/ρA)ρA ,E ↔ ⍳12 ρ,5 ↔ 1
Catenate	V,V	P,⍳2 ↔ 2 3 5 7 1 2 'T','HIS' ↔ 'THIS'
Index[3,4]	V[A] M[A;A] A[A;...;A]	P[2] ↔ 3 P[4 3 2 1] ↔ 7 5 3 2 E[1 3;3 2 1] ↔ 3 2 1 11 10 9 E[1;] ↔ 1 2 3 4 E[;1] ↔ 1 5 9 'ABCDEFGHIJKL'[E] ↔ ABCD EFGH IJKL
Index generator[3]	⍳S	First S integers ⍳4 ↔ 1 2 3 4 ⍳0 ↔ an empty vector
Index of[3]	V⍳A	Least index of A in V, or 1+ρV P⍳3 ↔ 2 P⍳E ↔ 5 1 2 5 3 5 4 5 5 5 5 5 4 5⍳4 ↔ 1
Take	V↑A	Take or drop |V[I] first (V[I]≥0) or last (V[I]<0) elements of coordinate I 2 3↑X ↔ ABC EFG
Drop	V↓A	¯2↓P ↔ 5 7
Grade up[3,5]	⍋A	The permutation which would order A (ascending or descending) ⍋3 5 3 2 ↔ 4 1 3 2
Grade down[3,5]	⍒A	⍒3 5 3 2 ↔ 2 1 3 4
Compress[5]	V/A	1 0 1 0/P ↔ 2 5 1 0 1 0/E ↔ 1 3 5 7 9 11 1 0 1/[1]E ↔ 1 2 3 4 ↔ 1 0 1⌿E 9 10 11 12
Expand[5]	V\A	1 0 1\⍳2 ↔ 1 0 2 1 0 1 1 1\X ↔ A BCD E FGH I JKL
Reverse[5]	⌽A	⌽X ↔ DCBA HGFE LKJI ⌽P ↔ 7 5 3 2 ⌽[1]X ↔ ⊖X ↔ IJKL EFGH ABCD
Rotate[5]	A⌽A	3⌽P ↔ 7 2 3 5 ↔ ¯1⌽P 1 0 ¯1⌽X ↔ BCDA EFGH LIJK
Transpose	V⍉A	Coordinate I of A becomes coordinate V[I] of result 1 1⍉E ↔ 1 6 11 2 1⍉X ↔ AEI BFJ CGK DHL
	⍉A	Transpose last two coordinates ⍉E ↔ 2 1⍉E
Membership	A∊A	ρV∊T ↔ ρV P∊⍳4 ↔ 1 1 0 0 E∊P ↔ 0 1 1 0 1 0 1 0 0 0 0 0
Decode	V⊥V	10⊥1 7 7 6 ↔ 1776 24 60 60⊥1 2 3 ↔ 3723
Encode	V⊤S	24 60 60⊤3723 ↔ 1 2 3 60 60⊤3723 ↔ 2 3
Deal[3]	S?S	W?Y ↔ Random deal of W elements from ⍳Y

Notes:

1 Restrictions on argument ranks are indicated by: *S* for scalar, *V* for vector, *M* for matrix, *A* for Any. Except as the first argument of *S*⍳*A* or *S*[*A*], a scalar may be used instead of a vector. A one-element array may replace any scalar.

2 Arrays used in examples: *P* ↔ 2 3 5 7   *E* ↔ 1 2 3 4 / 5 6 7 8 / 9 10 11 12   *X* ↔ ABCD / EFGH / IJKL

3 Function depends on index origin.

4 Elision of any index selects all along that coordinate.

5 The function is applied along the last coordinate; the symbols ⌿, ⍀, and ⊖ are equivalent to /, \, and ⌽, respectively, except that the function is applied along the first coordinate. If [*S*] appears after any of the symbols, the relevant coordinate is determined by the scalar *S*.

Figure 3

```
    ∇ R←AVERAGE V
[1]   R←(+/V)÷ρV
    ∇
```

Figure 4

```
    ∇ R←STATS X;SD;VAR;MEAN
[1]   R←MEAN,VAR,SD←(VAR←(+/(X-MEAN←AVERAGE X)*2)÷¯1+ρX)*
      0.5
    ∇
```

Figure 5

X to be all of those elements currently comprising X followed by the numbers 1.5 and 20.7, then X ← X, 1.5 20.7 *catenates* the constant vector 1.5 20.7 to X and *respecifies* X. The variable X is now a data object with 5 elements where X[1] is 2 X[4] is 1.5 and X[5] is 20.7. We may query the system as to the size (number of components) of X by use of the function denoted by the Greek letter Rho. Thus, ρX produces 5. The functions of *size* and *catenate* are summarized together with the rest of the mixed APL dyadic functions in Figure 3.

We will not here treat further the powerful functions of data manipulation illustrated there. However, we have now exposed the reader to a sufficient amount of detail in APL to understand Figure 4. This shows the listing of a user-written function, the name of which is AVERAGE. The first or *header line* of AVERAGE declares the syntax for that function, that is, it indicates that the explicit result will be called R and the vector of data to be averaged will be denoted by V. The line numbered [1] is the algorithm, and it is self explanatory, even at this point.

Figure 5 shows how AVERAGE is called within the function STAT to calculate the mean, variance, and standard deviation of a set of values. Here the variable names of MEAN, VAR, and SD refer to the result of the AVERAGE program and the calculated variance and standard deviation.

We do not illustrate the comparable programs in other languages; we leave to the reader the task of noting the coding compression achieved by APL. The APL array operations obviously provide both brevity and clarity in expression, and in that sense the programs may be thought of as somewhat self documenting.

The symbolic nature of APL makes it multilingual.

Evaluation of APL

In these pages we have only scratched the surface of APL. The availability of a powerful set of functions having a generality and a sense of uniformity in definition is important in providing capability to program complex algorithms. The extension of operations uniformly to strings of quantities or tables of numbers is a step forward in programming, because a great deal of computing in science, government, and business may be cast in terms of those data structures. Also it is important to relieve the computer user of the burden of bookkeeping and housekeeping operations in computer programming in higher level languages, particularly in an interactive environment.

Enthusiastic supporters of APL have claimed that rather than standing for either *A P*rogramming *L*anguage or *A*nother *P*rogramming *L*anguage, the initials APL stands for *A P*ermanent *L*anguage. APL was first conceived of as a means of communication; and it will have importance in that regard independent of the availability of APL on a terminal system. The heart of communicating, describing, or programming a process is to make clear what is to be done. In fact I might suggest that Ken Iverson and his colleagues meant APL to be a tool so that we all could program lucidly. □

An APL Bibliography

1. Abrams, P. S., **An Interpreter for "Iverson Notation"**. Stanford, Calif.: Computer Science Department, Stanford University, Tech. Report CS47, August 17, 1966.
2. Anscombe, F. J., **Use of Iverson's Language APL for Statistical Computing**. New Haven: Department of Statistics, Yale University, July, 1968. TR-4 (AD 672-557).
3. Berges, G. A. and F. W. Rust, **APL/MSU Reference Manual**. Bozeman, Montana: Department of Electrical Engineering, Montana State Univ., September 26, 1968.
4. Berry, P. C., **APL/1130 Primer**. IBM Corporation, 1968. (C20-1697-0).
5. Berry, P. C., **APL\360 Primer Student Text**. IBM Corporation, 1969. (C20-1702-0).
6. Breed, L. M. and R. H. Lathwell, "The Implementation of APL\360", **Interactive Systems for Applied Mathematics**. New York and London: Academic Press, 1968, pp. 390-399.
7. Calingaert, P., **Introduction to A Programming Language**. Chicago: Science Research Associates, field test edition, October, 1967.
8. Creveling, Cyrus J. (Ed.), **Experimental Use of A Programming Language (APL) at the Goddard Space Flight Center**. Greenbelt, Maryland: Goddard Space Flight Center, Report No. x-560-68-420, November, 1968.
9. Charmonman, S., S. Cabay and M. L. Louie-Byne, **Use of APL\360 in Numerical Analysis**. Edmonton, Alberta, Canada: Department of Computing Science, University of Alberta, December, 1967.
10. Falkoff, A. D. and K. E. Iverson, **APL\360 User's Manual**. Yorktown Heights, N.Y.: T. J. Watson Research Center, IBM Corporation, 1968.
11. Falkoff, A. D. and K. E. Iverson, "The APL 360 Terminal System", **Interactive Systems for Applied Mathematics**. New York and London: Academic Press, 1968, pp. 22-37. (Also Research Note RC 1922, October 16, 1967, T. J. Watson Research Center.)
12. Falkoff, A. D., K. E. Iverson and E. H. Sussenguth, "A Formal Description of System/360". **IBM Systems Journal**, III, No. 3 (1964), pp. 193-262.
13. Gilman, L. I. and A. J. Rose, **APL\360 An Interactive Approach**. IBM Corporation, 1969.
14. Hellerman, H., **Digital Computer System Principles**. New York: McGraw-Hill, 1967.
15. Iverson, I. E., "A Common Language for Hardware, Software and Applications". Eastern Joint Computer Conference, December, 1962, pp. 121-129 (RC 749).
16. Iverson, K. E., "The Description of Finite Sequential Processes", **Information Theory, 4th London Symposium**, Colin Cherry (Ed.). London: Butterworth's 1961.
17. Iverson, K. E., **Elementary Functions: An Algorithmic Treatment**. Chicago: Science Research Associates, 1966.
18. Iverson, K. E., **Formalism in Programming Language**. Yorktown Heights, N.Y.: T. J. Watson Research Center, IBM Corporation, July 2, 1963. (RC-992).
19. Iverson, K. E., **A Programming Language**. New York: John Wiley and Sons, Inc., 1962.
20. Iverson, K. E., "A Programming Language". Spring Joint Computer Conference, May, 1962, pp. 245-351.
21. Iverson, K. E., "Recent Applications of a Universal Programming Language". New York: IFIP Congress, May 24, 1965. (Also Research Note NC-511, T. J. Watson Research Center.)
22. Iverson, K. E., **The Role of Computers in Teaching**. Kingston, Ont., Canada: Queen's University, Queen's Papers on Pure and Applied Mathematics, No. 13, 1968. Also issued as **The Use of APL in Teaching**, IBM Corporation, 1969. (320-0996-0).
23. Kolsky, H. G., "Problem Formulation Using APL". **IBM Systems Journal**, 8, 3(1969), pp. 204-217.
24. Krueger, S. E. and T. P. McMurchie, **A Programming Language**. Chicago: Science Research Associates, 1968.
25. Lathwell, R. H., **APL\360: Operations Manual**. IBM Corporation, 1968.
26. Lathwell, R.H., **APL\360: System Generation and Library Maintenance**. IBM Corporation, 1968.
27. MacAuley, Thomas, **CAL/APL: Computer Aided Learning/A Programming Language**, Author's Manual. Costa Mesa, Calif.: Information Services and Computer Facility, Orange Coast Junior College.
28. Pakin, Sandra, **APL\360 Reference Manual**. Chicago: Science Research Associates, 1968. (No. 17-1).
29. Rose, A. J., **Teaching the APL\360 Terminal System**. Yorktown Heights, N.Y.: T. J. Watson Research Center, IBM Corporation, August 28, 1968. (RC 2184.)
30. Rose, A. J., Videotaped APL Course. IBM Corporation, 1967.
31. Simillie, K. W., **STATPACK II: An APL Statistical Package**. Edmonton, Alberta, Canada: Department of Computer Science, University of Alberta, Publication No. 17, February 1969.
32. Woodrum, L. J., "Internal Sorting with Minimal Comparing". **IBM Systems Journal**, 8, 3(1969) pp. 189-203.

Selected Bibliography for APL

[1] Berry, P.C., APL/360 Primer Student Text. IBM Corporation, 1969. (C20-1702-0).
An excellent introduction to the fundamentals of APL.

[2] Falkoff, A.D. and K.E. Iverson, APL/360 User's Manual. Yorktown Heights, N.Y.: T.J. Watson Research Center, IBM Corporation, 1968.

[3] Gilman, L.I. and A.J. Rose, APL/360 An Interactive Approach. IBM Corporation, 1969.
A textbook on APL (used in advanced undergraduate programming course at C-MU). Discusses some extensions to basic APL/360.

[4] Iverson, K.E., A Programming Language. New York: John Wiley and Sons, Inc., 1962.
The original definition of the notational scheme. Excellent in its own right, but not directly useful in learning one of the APL implementations.

[5] Pakin, Sandra, APL/360 Reference Manual. Chicago: Science Research Associates, 1968.
The definitive work on APL (as of 1968): explains each operator (with many examples). Note: this book is a reference manual, not a primer.

```
*  Documentation for APL/10 system at C-MU can be   *
*  found on the file  APL.DOC.  This file explains  *
*  the differences between APL/10 and APL/360 and   *
*  discusses the extensions implemented in APL/10,  *
*  as well as how to get onto the APL/10 system at  *
*  C-MU.                                            *
```

APL

Simple Examples and Problems

Write APL expressions to perform the following:

1. Remove all duplicate elements from a vector V, and call the resulting compressed vector RES.

2. Determine which vowels ('AEIOU') and how many of each appear in a given character string C.

3. Given a vector V, whose components are decimal integers, determine how many decimal places each component has.

Write APL functions to perform the following:

4. Write a function PRI to list the prime numbers that lie between the integers R and S, inclusive.

5. Let X be a vector whose components are arranged in ascending order. Define a function MERGE which will insert the components of a vector V so that the resulting vector R is still in ascending order.

6. Write a <u>one-line</u> function to determine if a square matrix M is symmetric or not and have it print out either 'THE MATRIX IS SYMMETRIC' or 'THE MATRIX IS NOT SYMMETRIC'.

7. Without using the <u>array</u> catenation extension of the ravel operator, write a function to:
   a. catenate a vector R rowwise to a given matrix M.
   b. catenate a vector C columnwise to a given matrix M.

   Do not assume that the lengths of R or C are proper.

APL

ANSWERS TO SIMPLE EXAMPLES AND PROBLEMS

```
1.    RES←((⍳V)=V⍳V)/V

2.    +/'AEIOU'∘.=C

3.    1+⌊10⍟|V

4.            ∇Z←R PRI S;T
      [1]     Z←(R≤T)/T←(2=+/[1]0=(⍳S)∘.|⍳S)/⍳S
              ∇


5.            ∇ X MERGE V
      [1]     R←R[⍋R←X,V]
              ∇


6.            ∇ SYM M
      [1]     'THE MATRIX IS ';(0∊M=⍉M)/'NOT ';'SYMMETRIC.'
              ∇


7.            ∇ M PLUSROW R
      [1]     (1 0+⍴M)⍴(,M),R,((⍴M)[2]⍴0)
      [2]     ⍝ NOTE--NUMERIC INPUT IS ASSUMED SINCE R IS
      [3]     ⍝         EXTENDED BY 0'S IF TOO SHORT.
              ∇

              ∇ M PLUSCOL C
      [1]     ⍉(1 0+⍴⍉M)⍴(,⍉M),C,((⍴M)[1]⍴0)
      [2]     ⍝ NOTE--NUMERIC INPUT IS ASSUMED SINCE C IS
      [3]     ⍝         EXTENDED BY 0'S IF TOO SHORT.
              ∇
```

APLSS\APL

TELETYPE SYSTEM MNEMONICS

TTY	*APL*	ALTERNATE TTY	TTY	*APL*
.AL	⍺	@A	.CB	⍀
.DE	⊥	@B	.CR	⊖
.DU	∩	@C	.CS	⌿
.FL	⌊	@D	.DQ	⌹
.EP	∊	@E	.GD	⍒
.US	_	@F	.GU	⍋
.DL	∇	@G	.IB	⌶
.LD	∆	@H	.IQ	⍇
.IO	⍳	@I	.LG	⍟
.SO	∘	@J	.NN	⍲
'	'	@K	.NR	⍱
.BX	⎕	@L	.OQ	⍈
.AB	|	@M	.OU	∪̸
.EN	⊤	@N	.PD	⍫
.LO	○	@O	.QD	⍔
*	*	@P	.QQ	⍞
?	?	@Q	.RV	⌽
.RO	⍴	@R	.TR	⍉
.CE	⌈	@S	.XQ	✖
.NT	~	@T	.ZA	A̲
.DA	↓	@U	.ZB	B̲
.UU	∪	@V	.ZC	C̲
.OM	⍵	@W	etc.	etc.
.LU	⊃	@X		
↑	↑	@Y		
.RU	⊂	@Z		

.DD	¨	"	⍝
.GE	≥	&	∧
.GO	→	#	×
.LE	≤	%	÷
.NE	≠	!	!
.NG	¯	$	$
.OR	∨	/	/
		(	(
		)	)
		etc.	etc.

Script
R. Fennel, F. Pollack

```
.R APL                                    Gets you into APL
CHARACTER SET..
TTY                                       type tty if you are at teletype
APL-OLS                                      or APL if at datel
TTY100) 19:11:16  8/19/71 [65,10]         You are now in APL
CLEAR WS
      3#4                                 entry is automatically indented
12                                        response is not
      X←3#4                               X is assigned the value of  3 times 4
      X
12                                        value of x typed out
      Y←-5                                y assigned -5
      X+Y                                 the sum of x plus y
7
      144E.NG2                            exponential form. .ng is special minus
1.44                                      for constants.  It is not an operator
      P←1 2 3 4                           assign the vector 1 2 3 4 to p
      P#P                                 multiply p by itself
1   4   9   16
      P#Y                                 scalar is apllied to all elements
-5  -10  -15  -20
      Q←'CATS'                            assign q a 4 element character vector
      Q
CATS
      3+4#5+2                             evàluation is from right to left
31                                          with no operator precedence
      X←3
      Y←4
      (X#Y)+4
16
      X#Y+4
24
      XY                                  the variable xy has not been defined
VALUE ERROR
      XY
      ^

      X←@I5                               index generator function
      X
1  2  3  4  5
      @I 0                                the vector of 0 elements

      Y←5-X                               all scalar functions extend to vectors
      Y
4  3  2  1  0
```

```
      X<Y                              result of relational operator is 0 or 1
1  1  Ø  Ø  Ø
      @O 1                             Pi times 1
3.141592
      3%2                              3 divided 2
1.5
      @O % 1 2                         pi divided by 1 2
3.141592  1.570796
      1@O 1                            sin 1
Ø.8414709
      2 @O1 2                          cos 1 2
Ø.5403022  -Ø.4161468
```

```
                                       Function Definition
      .DL Z←X F Y                      function header, result plus 2 parameters
[1]   Z←((X*2)+Y*2)*.5                 function body
[2]   .DL                              close of function
      3 F 4                            executing function
5                                      result
      P←7
      Q←(P+1)F P-1                     function call with expressions
      Q                                value assigned to q
1Ø
      4#3 F 4
2Ø
      @G B←G A                         g is signum function. ĦA and B are
[1]   B←(A>Ø)-A<Ø                      locals. function is monadic.
[2]   @G
      G 4                              monadic function call
1
      G .NG6
-1
      G X←-6                           assignments may be anywhere in statement
-1
      .DL H A                          same as G but no result
[1]   P←(A>Ø)-A<Ø
[2]   @G
      H .NG6
      P
-1
      Y←H .NG6
VALUE ERROR                            value error since function call returns
      Y←H -6                           no explicit result
     ↑
      .DL Z←FAC N;I                    FAC is factorial function
[1]   Z←1
[2]   I←Ø
[3]   L1:I←I+1                         L1 becomes 3 at entrance into function
[4]   .GO Ø# @I I>N                    L1 is local.
[5]   Z←Z#I
[6]   .GO L1
[7]   @G
      FAC 3
6
      FAC 5
12Ø
```

```
      T@HFAC←3 5                set to trace lines 3 and 5 of FAC
      X←FAC 3
FAC[3] 1                        Trace of FAC
FAC[5] 1
FAC[3] 2
FAC[5] 2
FAC[3] 3
FAC[5] 6
FAC[3] 4
      X
6
      T@HFAC←Ø                  set trace off

      @G G←M GCD N              Greatest common divisor
[1]   G←N
[2]   M←M @M N
[3]   .GO 4#M .NE Ø
[4]   [1]G←M                    correction of line 1
[2]   [4]N←G                    resume with line 4
[5]   [1.BX]                    display line 1
[1]   G←M
[1]   [.BX]                     display entire function
    .DL  G←M GCD N
[1]   G←M
[2]   M←M.AB N
[3]   .GO 4#M.NE Ø
[4]   N←G
    .DL
[5]   .GO 1                     enter new line
[6]   @G                        close of function. @g and .dl are the same
      36 GCD 44
4
      .DL GCD                   reopen definition
[6]   [4.1]M,N                  insert new line
[4.2] [.BX]                     display function
    .DL  G←M GCD N
[1]   G←M
[2]   M←M.AB N
[3]   .GO 4#M.NE Ø
[4]   N←G
[4.1] M,N
[5]   .GO 1
    .DL
[6]   .DL                       close function.
      36 GCD 44
8  36
4  8
4
```

```
       @G GCD[.BX]@G                      reopen,display, and close function
     .DL  G←M GCD N                       notice that when function is closed,
[1]    G←M                                the lines are automatically renumbered.
[2]    M←M.AB N
[3]    .GO 4#M.NE 0
[4]    N←G
[5]    M,N
[6]    .GO 1
     .DL
       .DL GCD
[7]    [@H5]                              delete line 5 of function
[5]    @G
       .DL Z←ABC X                        to demonstrate line editing
[1]    Z←(33#Q+(R#5)-6
[2]    [1.BX 8]                           edit line 1, print line and space in 8
[1]    Z←(33#Q+(R#5)-6
          /    1 /1                       / for delete, number for leave space.
[1]    Z←(3#Q)+(T#5)-6                    enter ) and t in proper place
[2]    .DL
       FAC 5                              FAC still defined
120
       )ERASE FAC                         Erase it

       FAC 5                              FAC no longer defined
SYNTAX ERROR
       FAC 5
      ↑
       )FNS                               List defined function in this workspace
ABC      F         G         GCD       H

       P←2 3 5 7                             assign p the vector 2 3 5 7
       @RP                                   dimension of p
4
       T←'OH MY'                             character vector
       @R T                                  dimension of t
5
       P,P                                   catenation of two numeric vectors
2  3  5  7  2  3  5  7
       T,T                                   catenation of two character vectors
OH MYOH MY
       P,T                                   catenation of numbers with characters
DOMAIN ERROR                                 not permitted
       P,T
      ↑


       N←5
       'NOTE: .IO';N;' IS ';.IO N            Mixed output
NOTE: .IO5 IS 1  2  3  4  5
```

```
      M←2 3@R 2 3 5 7 11 13         create matrix of dimension 2 3
      M

  2   3   5
  7  11  13
      2 4@R T                       reshape t into 2 4 matrix

OH M
YOH
      6@RM                          reshape matrix into vector
2  3  5  7  11  13
      •BX←P←,M                      ravel in row major order
2  3  5  7  11  13
      P[3]                          indexing
5
      P[1 3 5]                      indexing by a vector
2  5  11
      P[@I3]                        first 3 elements of p
2  3  5
      P[@RP]                        last element of P
13
      M[1;2]                        element in row 1 column 2 of m
3
      M[1;]                         row 1 of M
2  3  5
      M[1 1;3 2]                    rows 1 and 1, columns 3 2

  5   3
  5   3
      A←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      A[M]                          A matrix index produces a matrix result

BCE
GKM
      A[M[1 1;3 2]]

EC
EC
      M[1;]←15 3 12                 respecifying the first row of M
      M

 15   3  12
  7  11  13
      Q←3 1 5 2 4 6
      P[Q]
5  2  11  3  7  13
      Q[Q]
5  3  4  1  2  6
      P[3]
5
      )ORIGIN Ø                     set origin to 0
WAS 1
      P[3]                          fourth element of P
7
      P[Ø 1 2]                      first 3 elements of P
2  3  5
      @I5
Ø  1  2  3  4
      )ORIGIN 1
WAS Ø
```

```
      V←?3@R9                             get random 3 element vector whose elements
      M←?3 3@R9                           are less than 10. and 2 random matrices
      N←3\3\?3 3@R 9
      V
6  8  1
      M

   8   1   5
   8   4   8
   6   6   6
      N

   4   7   2
   9   6   4
   4   6   4
      M+N                                 sum element by element

  12   8   7
  17  10  12
  10  12  10
      M @D N                              Minimum

   4   1   2
   8   4   4
   4   6   4
      M<N                                 comparison(result 0 to 1)

  0  1  0
  1  1  0
  0  0  0
      +/V                                 sum reduction of v
 15
      #/V                                 product reduction
 48
      +/[1]M                              sum over first co-ordinate of m
 22  11  19
      +/[2]M                              sum over 2nd co-ordinate of M
 14  20  18
      +/M                                 sum over last co-ordinate of m
 14  20  18
      @S/M                                max over last co-ordinate of M
 8  8  6
```

```
      M+.#N                                 ordinary matrix inner product

  61   92   40
 100  128   64
 102  114   60
      M+.<N                                 inner product

 1  2  1
 1  1  0
 1  1  0
      M+.#V                                 +.# inner product with vector right
61  88  90                                  argument
      V
6  8  1
      V@J.#@I5                              Outer product (times)

   6   12   18   24   30
   8   16   24   32   40
   1    2    3    4    5
      V @J.<@I5                             Outer product with 1 2 3 4 5(less than)

  0  0  0  0  0
  0  0  0  0  0
  0  1  1  1  1
      .RO V@J.#M                            Outer product of rank 3
3  3  3
      Q←?10@R5                              random 10 element vector(1-5)
      Q
4  3  2  2  5  2  5  5  1  4
      +/[1]Q@J.=@I5                         Ith element of result is number of
1  3  1  2  3                               occurences of the value I in Q
      2 1.TR M                              ordinary transpose

   8   8   6
   1   4   6
   5   8   6
      .TR M                                 same as monadic transpose

   8   8   6
   1   4   6
   5   8   6
```

```
        Q
4   3   2   2   5   2   5   5   1   4
        3 •RV Q                          rotate q to left by 3
2   5   2   5   5   1   4   4   3   2
        •NG3•RV Q                        Rotate Q to right by 3
5   1   4   4   3   2   2   5   2   5
        -3 •RV Q                         negative of rotate Q to left by 3
-2  -5  -2  -5  -5  -1  -4  -4  -3  -2
        0 1 2•RV[1]M                     Rotate columns by different amounts

    8    4    6
    8    6    5
    6    1    8
        •NG2•RV[2]M                      rotation of all rows by 2 to right

    1    5    8
    4    8    8
    6    6    6
        1 2 3•RV M                       Rotation of rows

    1    5    8
    8    8    4
    6    6    6
        •RV Q                            Reversal of Q
4   1   5   5   2   5   2   2   3   4
        •RV[1]M                          Reversal of M along first co-ordinate

    6    6    6
    8    4    8
    8    1    5
        •RVM                             Reversal along last co-ordinate of M

    5    1    8
    8    4    8
    6    6    6
        U←Q>4
        U
0   0   0   0   1   0   1   1   0   0
        U/Q
5   5   5                                Compression of Q by logical vector U
        (@T U)/Q
4   3   2   2   2   1   4                compression by not U
        +/U/Q
15
```

```
      _1 Ø 1[1]M                  type-in error
SYNTAX ERROR
      1  Ø  1[1]M
     ↑
      [1•BX 9]                    editing of immediate line
      1 Ø 1[1]M
           1
      1 Ø 1/[1]M                  insert '/'

  8   1   5                       compression along first co-ordinate of M
  6   6   6
      (,M>5)/,M
8  8  8  6  6  6                  all elements of M which exceed 5
      V←1 Ø 1 Ø 1
      V\ɑI3                       expansion of iota 3
1  Ø  2  Ø  3
      V\M                         expansion along last co-ordinate of M

  8   Ø   1   Ø   5
  8   Ø   4   Ø   8
  6   Ø   6   Ø   6
      V\'ABC'                     expansion of character inserts blanks
A B C
      10ɑB 1 7 7 6                base 10 value of  1 7 7 6
1776
      _8 PB1 7 7 6                typing error
SYNTAX ERROR
      8 PB1 7  7  6
     ↑
      [1•BX7]
      8 PB1 7 7 6
        /1                        P should be ɑ
      8 ɑB1 7 7 6                 base 8 value of 1 7 7 6
1Ø22
      1Ø 1Ø 1Ø 1ØɑN1776           4 digit base 10 representation of 1776
1  7  7  6
      1Ø 1ØɑN1776                 2 dgit base 10 representation of 1776
7  6
      24 6Ø 6ØɑB1 3 25            mixed base value
38Ø5
      24 6Ø 6ØɑN38Ø5
1  3  25
      2ɑB1 Ø 1 1 Ø                base 2 value
22
```

```
      P
2  3  5  7  11  13
      P .IO 7                 least index of 7 in p
4
      P .IO 6                 6 not in p, result is 1+.ro P
7
      P .IO 4 5 6 7           least index of 4 5 6 7 in p
7  3  7  4
      Q←5 1 3 2 4
      R←Q.IO .IO@RQ
      R
2  4  3  5  1
      Q[R]
1  2  3  4  5
      A←'ABCDEFGHIJKLMN'
      A←A,'OPQRSTUVWXYZ'
      A
ABCDEFGHIJKLMNOPQRSTUVWXYZ
      A@I'CAT'               rank of c a t in alphabet
3  1  20
      J←A@I'CAT'
      A[J]
CAT
      3?5                    random choice of 3 out of 5 with no repeat
2  4  1
      6?3
RANGE ERROR
      6?3
     ↑
      X←8?8                  a random permutation vector
      X
7  1  3  2  8  6  5  4
      .GUX                   the grade up of X
2  4  3  8  7  6  1  5
      X[.GU X]               X in ascending order
1  2  3  4  5  6  7  8
      X[.GD X]               X in descending order
8  7  6  5  4  3  2  1
      U←A @E 'NOW IS THE TIME'  Membership
      (.BX←U)/A
0  0  0  0  1  0  0  1  1  0  0  0  1  1  1  0  0  0  1  1  0  0  1
      0  0
EHIMNOSTW
      (@I9)@E3 6 2 9
0  1  1  0  0  1  0  0  1
```

```
      .DL  Z←BIN N                                    BINOMIAL COEFFICIENT FUNCTION.
 [1]    Z←1
 [2]    LA:Z←(Z,0)+0,Z
 [3]    .GO LA#N.GE .RO Z
      .DL

      )FNS                                List of functions in workspace
ABC       BIN        ENTERTEXT              F          G          GCD        H
MULTDRILL
      )VARS                               List of variables in workspace
A         D          J          LA          M          N          P          Q          R
T         U          V          X           Y
      A←')FNS                             string containing two APL statements
)VARS'
      A
)FNS
)VARS
      B←@E A                              Execution of string. value of first printed
ABC       BIN        ENTERTEXT              F          G          GCD        H
MULTDRILL                                 Second assigned to B
      B
A         D          J          LA          M          N          P          Q          R
T         U          V          X           Y
      A←'BIN 3'
      B←@E A                              Execute string value returned in b
      B                                   print value of function call
1   3   3   1
      B←@N 'BIN'                          get  lines of function BIN
      B
.DLZ←BIN N
Z←1
LA:Z←(Z,0)+0,Z
.GOLA#N.GE.ROZ
.DL
      )ERASE BIN                          erase function

      BIN 3
SYNTAX ERROR
      BIN 3
      ↑
      @E B                                execute will redefine function

      BIN 3                               try it out
1   3   3   1
      INV←.DQ M                           get inverse of matrix
      M+.#INV                             result should be identity matrix

    1.000000E0        2.980232E-8      0.0
    0.0               1.000000E0       0.0
    0.0               2.980232E-8      1.000000E0
      )OFF HOLD                           sign off APL
TTY100) 20:52:05  8/19/71
CONNECTED    1:40:48 CPU TIME    0:00:17
```

BLISS

C. Geschke, C. Weinstock (Revised 8/75, R. Levin)

INTRODUCTION

BLISS-10 is a language specifically designed for writing software systems such as compilers and operating systems for the PDP-10. While much of the language is relatively "machine independent" and could be implemented on another machine, the PDP-10 was always present in our minds during the design; and as a result, BLISS-10 can be implemented very efficiently on the 10. This is probably not true for other machines.

We refer to BLISS-10 as an "implementation language." This phrase has become quite popular lately, but apparently does not have a uniform meaning. Hence, it is worthwhile to explain what we mean by the phrase and consequently what our objectives were in the language's design. To us the phrase "implementation language" connotes a higher level language suitable for writing production software; a truly successful implementation language would completely remove the need and/or desire to write in assembly language. Furthermore, to us, an implementation language need not be machine independent--in fact, for reasons of efficiency, it is unlikely to be.

Many reasons have been advanced for the use of a higher level language for implementing software. One of the most often mentioned is that of speeding up its production. This will undoubtedly occur, but it is one of the less important benefits, except insofar as it permits fewer, and better programmers to be used. Far more important, we believe, are the benefits of documentation, clarity, correctness, and modifiability. These were the most important goals in the design of BLISS-10.

Some people, when discussing the subject of implementation languages, have suggested that one of the existing languages, such as PL/I, or at most a derivative of one, should be used; they argue that there is already a proliferation of languages, so why add another. The only rational excuse for the creation of yet another new language is that existing languages are unsuitable for the specific applications in mind. In the sense that all languages are sufficient to model a Turing machine, any of the existing languages, LISP for example, would be adequate as an implementation language. However, this does not imply that each of these languages would be equally convenient. For example, FORTRAN can be used to write list processing programs, but the lack of recursion coupled with the requirement that the programmer code his own primitive list manipulations and storage control makes FORTRAN vastly inferior to, say, LISP for this type of programming.

What, then, are the characteristics of systems programming which should be reflected in a language especially suited for the purpose? Ignoring machine dependent features (such as a specific interrupt structure) and recognizing that all differences in such programming characteristics are only one of degree, three features of systems programming stand out:

1. Data structures. In no other type of programming does the variety of data structures nor the diversity of optimal representations occur.
2. Control structures. Parallelism and time are intrinsic parts of the programming system problem.*
3. Frequently, systems programs cannot presume the existence of large support routines (for dynamic storage allocation, for example).

* Of course, parallelism and time are intrinsic to real time programming as well.

These are the principal characteristics which the design of BLISS-10 attempts to address. For example, taking point (3), the language was designed in such a way that no system support is presumed or needed, even though, for example, dynamic storage allocation is provided. Thus, code generated by the compiler can be executed directly on a "bare" machine. Another example, taking point (1), is the data structure definition facility. BLISS contains no implicit data structures (and hence no presumed representations for structures), but rather provides a method for defining a representation by giving the explicit accessing algorithm.

CMU I/O and Other Packages:

Several packabes of useful subroutines have been developed for BLISS-10. Here is a list of these packages, with pointers to relevant documentation:

BLILIB:

This is the so-called BLISS library, which contains i/o conversion routines, file access routines, storage allocation routines, and other facilities.

Documentation:	SYS:BLILIB.DOC
Maintainer:	R. Johnsson

TIMER:

A package which can be loaded with your BLISS-10 to provide statistics on the run-time of routines in your BLISS-10 program. Extremely useful in the design-implementation cycle of an efficient programming system.

Documentation:	TIM.DOC
Implementor:	J. Newcomer

POOMAS:

"Poor-Mans-Simulation-Package." An adjoint to BLISS-10 of the same flavor as the union of SIMULA and ALGOL.

Documentation:	POOMAS.DOC
Implementor:	A. Lunde

SIX12:

A high level debugging package. Since it knows about the Bliss-10 run time enviroment it is useful in interactive Bliss deburring.

Documentation:	SIX12.DOC
Implementors:	C. Weinstock W. Wulf T. Lane

REFERENCES

[1] Wulf, Russell, Habermann, Geschke, Apperson, Wile, Brender, "BLISS Reference Manual," Computer Science Department Report, CMU, 1970.

[2] Wulf, Russell, Habermann, "BLISS: A Language for Systems Programming," DECUS Proceedings, Spring, 1970.

[3] Wile, Geschke, "Efficient Data Accessing in the Programming Language BLISS," SIGPLAN Conf. on Data Structures in Programming Languages, SIGPLAN Notices, February, 1971.

[4] Wulf, Geschke, Wile, Apperson, "Reflections on a Systems Programming Language," SIGPLAN Conf. on Implementation Languages, SIGPLAN Notices, October, 1971.

[5] Wulf, Russell, Habermann, "BLISS: A Language for Systems Programming," C.A.C.M. (to be published).

Some fairly extensive examples have been prepared as an appendex to the BLISS-10 Reference Manual.

SIMPLE EXAMPLES

```
1)  ! find index of first space in a line
    ! image of 80 characters (one per word)
    ! index = -1 implies none found

        index ← incr j from ∅ to 79 do
                    if . line [.j] eql #4∅ then
                            exitloop .j;

2)  ! find last item of simple list
        link ← . beginning of linked list;
        while .. link neg ∅ do link ← ..link;
    ! link contains address of last item

3)  ! add the first N numbers
        sum ← ∅;
        incr j from 1 to .n do sum ← .sum+.j;

4)  ! routine to compute factorial
        routine factorial (n) =
            if .n eql ∅
                then 1
                else .n* factorial (.n-1);
```

THE FOLLOWING IS AN EXAMPLE OF A TERMINAL SESSION USING BLISS-10. COMMENTS ARE DISTINGUISHED FROM ACTUAL MACHINE INTERACTION BY BEING ENCLOSED IN ----'ED LINES. SINCE BLISS-10 HAS NO BUILT-IN I/O FACILITIES, THIS EXAMPLE USES A FILE CALLED TTIO.BLI WHICH CONTAINS SOME 'BARE BONES' TERMINAL SUPPORT. MORE EXTENSIVE AND CONVENIENT I/O FACILITIES CAN BE FOUND IN THE BLISS LIBRARY (SYS:BLILIB.REL).

```
.TYPE TTIO.BLI


!       PDP-10 I/O FACILITIES FOR TTY


MACHOP TTCALL = #51;                          !UUO FOR TTY OPERATIONS

MACRO   INC = (REGISTER Q; TTCALL(4,Q); .Q)$,
        OUTC(Z) = (REGISTER Q; Q←(Z); TTCALL(1,Q))$,
        OUTSA(Z) = TTCALL(3,Z)$,
        OUTS(Z) = OUTSA(PLIT ASCIZ Z)$,
        OUTM(C,N) = DECR I FROM (N)-1 TO 0 DO OUTC(C)$,
        CR = OUTC(#15),  LF = OUTC(#12)$,  NULL = OUTC(0)$,
        CRLF = OUTS('?M?J?0?0')$,
        TAB = OUTC(#11)$;


!       GENERAL NUMERIC CONVERSION
!               OUTPUTS .NUM IN RADIX .BASE IN A FIELD AT
!               LEAST .REQD CHARACTERS WIDE

ROUTINE OUTN(NUM,BASE,REQD) =
  BEGIN OWN N,B,RD,T;
    ROUTINE XN =
      BEGIN LOCAL R;
        IF .N EQL 0 THEN RETURN OUTM("0",.RD-.T);
        R←.N MOD .B; N←.N/.B; T←.T+1; XN();
        OUTC(.R+"0")
      END;

    IF .NUM LSS 0 THEN OUTC("-");
    B←.BASE; RD←.REQD; T←0; N←ABS(.NUM); XN()
  END;


!       COMMON USES FOR 'OUTN'


MACRO   OUTD(Z) = OUTN(Z,10,1)$,
        OUTO(Z) = OUTN(Z,8,1)$,
        OUTDR(Z,N) = OUTN(Z,10,N)$,
        OUTOR(Z,N) = OUTN(Z,8,N)$;
```

NOW WE WILL BUILD A PROGRAM TO PRINT AT THE TERMINAL THE FACTORIALS FROM 0 TO 12. WE HAVE ALREADY CREATED THE FILE FACT.BLI USING TECO. ITS CONTENTS ARE:

```
.TYPE FACT.BLI


MODULE FACT(STACK) =

BEGIN

REQUIRE TTIO.BLI;                    !GET DECLARATIONS FROM TTIO.BLI

ROUTINE FACTORIAL(N) =
  IF .N EQL 0 THEN 1 ELSE .N*FACTORIAL(.N-1);


!       MAIN PROGRAM

CRLF; TAB; OUTS('N'); TAB; OUTS('N!'); CRLF; CRLF;
INCR I FROM 0 TO 12 DO
  BEGIN
    TAB;
    OUTD(.I);
    TAB;
    OUTD(FACTORIAL(.I));
    CRLF;
  END


END

ELUDOM
```

NOW WE ARE READY TO COMPILE THE PROGRAM. BLISS-10 ACCEPTS THE STANDARD DEC COMMAND STRING ALONG WITH A LARGE NUMBER OF OPTIONAL (AND DEFAULTED) SWITCHES WHICH ARE DESCRIBED IN THE MANUAL. IN THIS EXAMPLE WE ARE NOT GOING TO USE ANY OF THE CCL COMMANDS ALTHOUGH THE CMU MONITOR DOES RECOGNIZE THE .BLI EXTENSION AND WILL HANDLE BLISS-10 FILES.

THE COMMAND STRING WILL PRODUCE A FILE NAMED FACT.REL.

```
.R BLISS
*FACT,←FACT

MODULE LENGTH: 131+16

[BLSNED NO ERRORS DETECTED]

*↑C
```

NOW WE ARE READY TO LOAD AND EXECUTE THE PROGRAM.

```
.LOAD FACT
LOADING

LOADER 1+1K CORE

EXIT

.START
        N          N!

        0          1
        1          1
        2          2
        3          6
        4          24
        5          120
        6          720
        7          5040
        8          40320
        9          362880
        10         3628800
        11         39916800
        12         479001600

EXIT

.
```

LISP

Crispin Perdue
Mike Rychener
Don Waterman

The following quote from the introduction to Weissman's Primer [5] will serve to introduce the language:

"LISP is an unusual language in that it is both a formal mathematical language, and (with extensions) a convenient programming language. As a formal mathematical language, it is founded upon a particular part of mathematical logic known as recursive function theory. As a programming language, LISP is concerned primarily with the computer processing of symbolic data rather than numeric data.

"From childhood we are exposed to numbers and to ways of processing numerical data, such as basic arithmetic and solutions to algebraic equations. This exposure is based upon a well-established and rigorously formalized science of dealing with numbers. We are also exposed to symbolic data -- such as names, labels, and words -- and to ways of processing such data when we sort, alphabetize, file, or give and take directions. Yet the processing of symbolic data is not a well-established science. In learning an algebraic programming language, such as FORTRAN or ALGOL, we call upon our experience with numbers to help us understand the structure and meaning (syntax and semantics) of the language.

"In learning a symbolic programming language such as LISP, however, we cannot call upon our experience, because the formalism of symbolic data processing is not part of this experience. Thus, we have the added task of learning a basic set of formal skills for representing and manipulating symbolic data before we can study the syntax and semantics of the LISP 1.5 programming language.

"LISP is designed to allow symbolic expressions of arbitrary complexity to be evaluated by a computer. To achieve a thorough understanding of the meaning, structure, construction, and evaluation of symbolic expressions, is to learn how to program in LISP."

The current version of LISP at CMU is an accretion onto Stanford LISP 1.6, in several layers. Stanford 1.6 itself is a slightly more usable version of LISP than the 1.5 version described by Weissman [5]. The differences are easily grasped, so that little unlearning is needed to go from a textbook LISP to a running computer implementation. The Primer is essential for the beginner, since the reference manual [3] is organized for reference rather than for assimilating essential language features. LISP 1.6 was improved considerably by the UCI LISP team [1]. Particularly helpful to the user are the powerful debugging and editing facilities, modelled after Inter-Lisp. The UCI LISP version has a number of new basic LISP functions also. The most useful parts of this augmented system are introduced tutorially in [2]. Finally, there is the LISPX [4] extension to LISP, which contains useful input-output functions, machine-specific functions (for the local CMU environment), and other power and convenience features.

LISP at CMU is self-documenting. That is, a user can access portions of the reference manual on-line, by using the HELP function. There is also an index available through the HELP function, which can point the user towards other HELP messages.

HIGHLY SELECTED REFERENCES

[1] Bobrow, R., R. Burton, and D. Lewis, UCI LISP Manual. U. of Calif. at Irvine, 1973.

[2] Perdue, Crispin, User's Introduction to UCI LISP. CMU CSD Tech. Report, 1974.

[3] Quam, Lynn, Stanford LISP 1.6 Manual. Stanford AI Memo, 1969.

[4] Stickel, Mark, LISPX. Internal document, CMU, 1976.

[5] Weissman, Clark, LISP 1.5 Primer. Dickenson Publishing Co., 1967.

LISP SCRIPT

What follows is an absolutely minimal demonstration of the use of UCI LISP and some important features, especially input/output using files. Comments here are like those in actual LISP source files: they begin with ~ (<control Z>, or SOS ?4) and go to the end of the line.

```
.R LISP
Type (help help) if you need it.

<1>(core)                     ~ How to increase your storage allocations.
                              ~ LISP prompts with <number>.
15872                         ~ Find out how much you have, in words.
<2>(core 22000)               ~ Increase the size to 22K; invokes allocation.
ALLOC (Y OR N) Y              ~ DO NOT PUT CRLF OR ANYTHING ELSE AFTER "Y"
FULL WORD SP. =               ~ For no change, type a blank.  NO CRLF!
BIN. PROG. SP. =              ~ Again no change.
REG. PDL. = 1000              ~ Terminate with a blank.  DO NOT HIT RETURN!
SPEC. PDL. = 1000             ~ Remainder of the extra is assigned to FREE SPACE.
FREE STG EXHAUSTED
FULL WORD SPACE EXHAUSTED
9255 FREE STG, 1241 FULL WORDS AVAILABLE
                              ~ Now conversation begins with LISP itself.
```

```
<3>(recordfile (lispin.rec))      ~ Send all termiral interactions to a file.

RECORD FILE DSK: (LISPIN . REC) OPENED 02-AUG-77 17:20:49
NIL
<4>10                             ~ The value of a number is itself.

10
<5>(quote (a b c))                ~ Evaluating a quoted expression removes
                                  ~ the QUOTE, but does not evaluate the
(A B C)                           ~ expression -- returns it unevaluated.
<6>(car (quote (a b c)))          ~ The basic CAR operation.
                                  ~ Note that LISP makes everything upper case.
A
<7>(cdr (quote (a b c]            ~ The square bracket closes all parentheses.

(B C)                             ~ The CDR.
<8>(cons 'a '(b c))               ~ The READ routine recognizes the "single
                                  ~ quote" as an "abbreviation" for QUOTE.
(A B C)
<9>(print ''a)                    ~ READ puts in the QUOTE's before EVAL gets to look
                                  ~ at the SEXPRESSION at all.
(QUOTE A)
<10>((lambda (x) (times x x)) 12)          ~ LAMBDA expressions a la Weissman.
                                           ~ Notice the different format here.
144
<11>((lambda (x y) (plus (times x x) (times y y))) 3 4)

25                                         ~ A LAMBDA expression of two variables.
<12>(de square (x) (times x x]             ~ The LISP function DE has the
                                           ~ side-effect of defining the function.
SQUARE                                     ~ It returns the atom naming the function.
<13>(pp square)                            ~ PP (alias GRINDEF) is for pretty-printing
                                           ~ function definitions.
(DEFPROP SQUARE                            ~ This is the DEFPROP format.
 (LAMBDA ((X) (TIMES X X))
EXPR)                                      ~ A function read in using this format will
NIL                                        ~ become defined, or perhaps redefined.
<14>(square 32)

1024                                       ~ It works.
<15>(de substitute (what for in)
>(cond ((equal for in) what)
>((atom in) in)
>(t (cons (substitute (car in))(substitute (cdr in]

SUBSTITUTE
```

```
<16>(pp substitute)                        ~ Print it out pretty.

(DEFPROP SUBSTITUTE
 (LAMBDA(WHAT FOR IN)
  (COND ((EQUAL FOR IN) WHAT)
        ((ATOM IN) IN)
        (T (CONS (SUBSTITUTE (CAR IN)) (SUBSTITUTE (CDR IN))))))
EXPR)

NIL
<17>(substitute 'x 'a '(a b c))            ~ Substitute X for A in (A B C).

TOO FEW ARGUMENTS SUPPLIED - APPLY

(SUBSTITUTE BROKEN)                        ~ LISP is unhappy with my function.
:what                                      ~ I can look at all the current values of things.
                                           ~ The BREAK package prompts with ":".
X
:bkv 3                         ~ I can also do a "backtrace" of all the expressions
                               ~ whose evaluation is still in progress.  The number,
                               ~ e.g. 3, limits how far back the backtrace will go.
(SUBSTITUTE (CAR IN))                      ~ Most recently entered expression prints first.
(CONS \#\ (SUBSTITUTE (CDR IN)))   ~ \#\ means "the previously printed expression".
(COND ((EQUAL FOR IN) WHAT)(T \#\))        ~ We see what called what.
   WHAT = X                                ~ Variable bindings are shown on entry
   FOR = A                                 ~ to a function.
   IN = (A B C)
(SUBSTITUTE (QUOTE X) (QUOTE A) (QUOTE (A B C)))

:↑                                         ~ Get "out of the break".

<18>(editf substitute)                     ~ Edit the function.
EDIT
#pp                                        ~ PP is also an editor command.
(LAMBDA(WHAT FOR IN)
 (COND ((EQUAL FOR IN) WHAT)
       ((ATOM IN) IN)
       (T (CONS (SUBSTITUTE (CAR IN))(SUBSTITUTE (CDR IN))))))

#f substitute                  ~ Find the first expression beginning "SUBSTITUTE".

(SUBSTITUTE (CAR IN))
#(insert what for before car)              ~ Do an insertion.
(SUBSTITUTE WHAT FOR (CAR IN))
#f substitute                              ~ Find the next "SUBSTITUTE".
```

```
(SUBSTITUTE (CDR IN))
#(INSERT WHAT FOR BEFORE (CDR IN))   ~ Insert again.

(SUBSTITUTE WHAT FOR (CDR IN))
#0 p                                 ~ 0 means "go up one level of parentheses.
                                     ~ P uses "&" ("etc.") for deeply nested lists.
(CONS (SUBSTITUTE WHAT FOR &)(SUBSTITUTE WHAT FOR &))
#↑ pp                                ~ ↑ (uparrow) means go back to the top,
                                     ~ the whole expression being edited.
(LAMBDA (WHAT FOR IN)
 (COND ((EQUAL FOR IN) WHAT)
       ((ATOM IN) IN)
       (T
        (CONS (SUBSTITUTE WHAT FOR (CAR IN))
              (SUBSTITUTE WHAT FOR (CDR IN))))))

#ok                                  ~ Exit the editor by saying OK.

SUBSTITUTE                           ~ The editor returns a value.
<19>(substitute 'x 'a '(a b c))      ~ Try again.

(X B C)                              ~ OK.
<20>(setq list '((a b) (d a b) (a b c)(d (a b))))

((A B) (D A B) (A B C) (D (A B)))
<21>(substitute '(x) '(a b) list)    ~ A more elaborate test.

((X) (D X) (A B C) (D (X)))          ~ Correct, but LOOK CLOSELY.
<22>(setq allfns '(substitute square))  ~ Get ready to save the definitions.

(SUBSTITUTE SQUARE)
<23>(help sort)                      ~ In general, the list of functions
                                     ~ is much longer and needs to be sorted.
 - - - - (SORT BASIC) - - - -        ~ This is the on-line help facility.
(SORT DATA COMPAREFN)  [SUBR]
SORT destructively sorts the list DATA using COMPAREFN as
a binary comparison function.  (COMPAREFN X Y) should
return something non-NIL if X can precede Y in sorted
order, NIL if Y must precede X.  If COMPAREFN is NIL,
LEXORDER will be used; if COMPAREFN is T, LEXORDERCAR will
be used.  Pointers to the head of DATA will not generally
continue to do so after a SORT.  The value returned is a
pointer to the new head of the list.

 - - - - (SORT SEE) - - - -
WANT TO SEE THE MESSAGE? >y
```

```
LEXORDER LEXORDERCAR

NIL
<24>(setq allfns (sort allfns 'lexorder))  ~ Here, lexorder is used as
                                           ~ the comparison, to alphabetize.
(SQUARE SUBSTITUTE)
<25>(dskout (allfns.lsp) allfns)           ~ ALLFNS.LSP is the file name.
                               ~ (DSKOUT (ALLFNS.LSP) '(SQUARE SUBSTITUTE))
NIL                            ~ would not work.  DSKOUT requires that you use an
                               ~ atom whose value is a list of functions.  Value
                               ~ of NIL returned by DSKOUT is OK.
<27>(recordfile)               ~ Close the file recording terminal interactions.
RECORD FILE DSK: (LISPIN . REC) CLOSED 02-AUG-77 17:31:24
↑C                                         ~ Exit to monitor.
.TY ALLFNS.LSP                             ~ Now to type the file, for proof.

(DEFPROP ALLFNS                            ~ Notice the DEFPROP format, just like PP.
 (ALLFNS SQUARE SUBSTITUTE)
VALUE)

(DEFPROP SQUARE
 (LAMBDA (X) (TIMES X X))
EXPR)

(DEFPROP SUBSTITUTE
 (LAMBDA(WHAT FOR IN)
  (COND ((EQUAL FOR IN) WHAT)
        ((ATOM IN) IN)
        (T (CONS (SUBSTITUTE WHAT FOR (CAR IN)) (SUBSTITUTE WHAT FOR (CDR IN))))))
EXPR)
.R LISP                                    ~ Now to run LISP again.

Type (help help) if you need it.
<1>(dskin (allfns.lsp))                    ~ DSKIN will read the file from disk.
                                           ~ NOTE:  for a file with no extension
ALLFNS                                     ~ say, e.g. (DSKIN ALLFNS).
SQUARE
SUBSTITUTE                                 ~ Names of things defined are printed out.
FILES-LOADED                               ~ Just a message.
<2>(square 111)

12321
<3>(substitute 'a 'b '(a b c))

(A A C)
<4>↑C                          ~ Amazing.
```

NUTSHELL INTRODUCTION TO UCI LISP

The Editor

When editing something, i.e., a list structure, in UCI LISP, one imagines it as consisting of expressions and subexpressions. A subexpression is simply a legitimate expression which happens to be part of another expression. In the list structure (A (B C) (D (E F))), some of the subexpressions are A, (B C), and (E F). A list is composed of elements, sometimes called top-level subexpressions. The elements of the example list above are precisely A, (B C), and (D (E F)). Of course the list elements themselves may have elements.

The attention of the editor is always centered on some subexpression of the structure being edited, possibly that structure itself. The expression on which the attention is focussed is known as the current expression. The elements of the current expression can be thought of as being numbered. If the current expression is (A (B C) (D (E F))), A is number 1, (B C) is number 2, and (D (E F)) is number 3. The attention of the editor can be moved to an element of the current expression by typing the appropriate number to the editor. The attention of the editor can be moved to the expression which contains the current expression as an element by typing 0. It is possible to shift the editor's attention back up to the entire expression being edited by the command ↑ (uparrow).

These numbers serve as commands to the editor. The current expression can be printed by the commands P and PP. P prints only the elements of the current expression and their elements. If an element of an element of the current expression is not an atom, it prints as merely "&". By default, the editor prints the current expression with P after executing each line of commands. Suppose the current expression is (A (B C) (D (E F))) and we are conversing with the editor.

```
#P

(A (B C) (D &))

#3

(D (E F))
```

The last element of the current expression can also be referred to as -1, the second to last as -2, etc..

It is also possible to move the attention of the editor to any subexpression of the current expression, not necessarily an ELEMENT, by using the F command with the name of the first element of the list. Actually, this only works if the first element of the list (subexpression) is an atom. If we are editing the list in the previous example:

```
#F D                    ~ The find command.

(D (E F))
#0

(A (B C) (D &))
#F E

(E F)
```

The F command does not limit itself to the current expression unless the current expression is the entire structure being edited. The F command searches forward through the entire structure being edited, and it searches in the order in which the expression appears when printed out, somewhat as though it were SOS or TECO. It begins, however, with the current expression, and only continues through the entire expression being edited if it does not find anything appropriate in the current expression. The search only looks at that part of the total expression which is to the right of the current expression. The F command always moves forward; that is, it never leaves the current expression the same. If the current expression is (A (B C) (D (B E))), editing might look like this:

```
#F D

(D (B E))
#F B

(B E)
```

The same expression could be found by:

```
#F B

(B C)
#F B

(B E)
```

At this point we can begin to explain the commands that actually change the list structure, e.g. your function definition. Take the INSERT command as an example.

(INSERT e1 . . . em <command> $)
where <command> := BEFORE or AFTER

The e1 . . . em indicates a sequence of expressions to be used, in this case inserted as is into the list structure. The $ indicates that what the user types in that part of the command is used as a "location specification". A location specification is very much like a sequence of F commands and numeric commands typed to the editor to change its focus of attention. In general, it is not necessary to type F <this> F <that>, etc., but only <this> <that> unless this or that is a number and therefore liable to be interpreted as a numeric command. The INSERT command as well as the others described here return the editor's attention to its original location after the operation is performed. For examples, look again at the INSERT commands in the script.

It is easy to imagine a current expression like (A B D) into which we wish to insert a C. It is perfectly legal to say (INSERT C AFTER B), even though B is not the CAR of a list. The only difference is that in this case the expression found is B itself, just as would be expected. CAUTION: The editor has many more commands than are mentioned here. If something in a location specification is also a command, the editor will use it as a command in the location specification unless it is preceded by "F". This is usually no problem.

The commands INSERT, DELETE, EXTRACT, EMBED, MOVE, and CHANGE form a useful set. Here follow statements of the syntax of the rest of these commands, illustrated by examples. We will take the initial current expression to be (BIM BAM (ZAM (GLUP) DING)) in each example unless otherwise noted.

Using INSERT, (INSERT HOWDY AFTER DING) would result in (BIM BAM (ZAM (GLUP) DING HOWDY)). The same effect could be achieved by (INSERT HOWDY AFTER ZAM 2).

(CHANGE $ TO e1 . . . em)

The CHANGE command causes the thing found by the location specification to be replaced by e1 to em. (CHANGE 3 TO (CRUNCH OOF) UGH) would result in (BIM BAM (CRUNCH OOF) UGH).

(DELETE $)

The DELETE command causes the expression found to be deleted. (DELETE DING) or (DELETE 2 3) would result in (BIM BAM (ZAM (GLUP))).

(EXTRACT $1 FROM $2)

The extract command causes an expression to be replaced by one of its subexpressions. (EXTRACT GLUP FROM ZAM) would result in (BIM BAM (GLUP)).

(EMBED $ IN e1 . . . em)

The EMBED command is slightly different from the others. The location specification operates in the usual way, and copies of the expression specified are made to be subexpressions of e1 to em. There is a special, but simple syntax for indicating where the editor is to put the copies of the expression found. Wherever the atom "*" appears in e1 to em, the expression found will appear in the result. e1 to em end up occupying the same position that the located expression occupied before the command. (EMBED BAM IN (*)) would result in (BIM (BAM) (ZAM (GLUP) DING)). (EMBED BAM IN (HELLO *)(WHAT (IS THIS *))) would result in (BIM (HELLO BAM)(WHAT (IS THIS BAM)) (ZAM (GLUP) DING)).

(MOVE $1 TO "before or after" S2)

The MOVE command contains 2 location specifications. S1 is performed and then, without losing its grip on the expression found, the editor moves its attention back to the initial current expression, does S2, and inserts the expression found by $1 at that point, removing it from its original position.

THRU

Sometimes it is useful to be able to deal with segments of a list, i.e. a sequence of elements rather than just one element. Consider the plausible example where the user has created (APPEND CAR A B) and obviously (APPEND (CAR A) B) was intended. It would be nice to do an EMBED command, but how is it to be done? The solution is to say (EMBED (CAR THRU A) IN (*)), resulting in (APPEND (CAR A) B). The general format is (S1 THRU S2). S2 is somewhat special, because it fails unless it finds an expression which is a element of the same list as the expression found by $1. Indeed it starts (and ends) its search with the list that the expression found by $1 is an element of. For example if the initial current expression is (A (B E)(G B E J)), we could say (MOVE (B B THRU E) TO BEFORE G), and we would get (A (B E) B E (G J)). (MOVE B (B THRU E) TO BEFORE G) would have the same effect, and so would (MOVE G (2 THRU 3) TO BEFORE G), which is a little more natural perhaps.

UNDO

One of the most remarkable features of the editor is that all of the commands, like INSERT, that actually make changes, can be UNDONE! To undo a command, say to the editor, UNDO <command>, for example,

```
#UNDO INSERT
```

and the most recent command of that type will be undone for you. Even the UNDO command can be undone by saying,

```
#UNDO UNDO
```

I know it's amazing, but it works just fine, and the moral to the story is TRY THE EDITOR. You just can't do yourself no harm (hardly). Honest.

DEBUGGING LISP

A word or two now on debugging. Backtracing has been demonstrated in the script. It is also possible to do tracing of function calls. When a function is being traced, its name is printed as it is entered, and the values of its parameters are displayed. When it is through, its name is printed again with the value it has returned. All of this is done in a graphically pleasing fashion.

To cause a function to be traced, say to LISP, (TRACE <function name>). To stop LISP from tracing it any more, say (UNTRACE <function name>). To stop LISP from tracing all thirty seven of the functions you have been tracing, say (UNTRACE). <function name> should NOT be QUOTE'd.

Another useful function, something like TRACE, is BREAK, perhaps an unlikely name for a nice feature. When a function is BROKEN, execution stops and you converse with LISP in much the same fashion as if there had been an error, as in the script. Most important, you can do all of the same kinds of poking around at things. If you BREAK a function on purpose, though, you can cause execution to CONTINUE by saying OK. For example,

```
:OK
```

From time to time you may get LISP into an infinite loop. To stop it type <control C>. Then to the monitor type REE<return> and finally either ↑B (<ctrl B>) to stop execution in the middle or ↑Z (<ctrl Z>) to forget the whole thing without losing LISP. For example,

```
↑C
.REE
↑Z

<5>
```

SAMPLE PROBLEMS

Write LISP functions for the following:

1. Determine whether an atom is in a list.
   e.g. (member 'b '(a b c)) = T
        (member 'x '(a b c)) = NIL

(member 'a '(b (a b) c) = NIL

2. Produce a tale (list of dotted pairs) given two lists, one of the references, and the other of the values.
   e.g. (pair '(one two three) '(1 2 3)) = ((one . 1) (two . 2) (three . 3))
   (pair '(plane sub) '(B47 Thresher)) = ((plane . B47) (sub . Thresher))

3. Append one list onto another.
   e.g. (append '(a b c) '(d e f)) = (a b c d e f)
   (append '((a b) c (d (e))) '((a))) = ((a b) c (d (e)) (a))

4. Delete an element from a list.
   e.g. (delete 'y '(x y z)) = (x z)
   (delete 'x '((u v) x y)) = ((u v) y)

5. Reverse a list. (Hint: use append.)
   e.g. (reverse '(a b c)) = (c b a)
   (reverse '(a (b c) d)) = (d (b c) a)

6. Produce a list of all the atoms which are in either of two lists.
   e.g. (union '(u v w) '(w x y)) = (u v w x y)
   (union '(a b c) '(b c d)) = (a b c d)
   (union '(a b c) '(a b c)) = (a b c)

7. Produce a list of all the atoms in common to two lists.
   e.g. (intersection '(a b c) '(b c d)) = (b c)
   (intersection '(a b c) '(a b c)) = (a b c)
   (intersection '(a b c) '(d e f)) = NIL

8. Find the last element on a list.
   e.g. (last '(a b c)) = c
   (last '((a b) (c))) = (c)

9. Reverse all levels of a list.
   e.g. (superreverse '(a b (c d))) = ((d c) b a)
   (superreverse '((u v) ((x z) y))) = ((y (z x)) (v u))

10. Determine whether a given atomic symbol is some part of an S-expression.
   e.g. (part 'a 'a) = T
   (part 'a '(x . (y . a))) = T
   (part 'a '(u v (w . x) z)) = NIL

L☆

G. Robertson

L☆ is a system for constructing software systems, which has been developed at CMU by A. Newell, D. McCracken, and G. Robertson. There are two up-to-date versions of the system; one on the PDP10 called L☆(I), and one on C.mmp called L☆C.(D). The system has gone through many iterations on several machines: L☆(I) is the tenth system to be designed for the PDP10 and the seventh to become a running system, and L☆C.(D) is the fourth running system on C.mmp. There have also been four running versions on stand-alone PDP11's, the most current version being L☆11(H). A running system on 360 TSS also exists, L☆360. Also, versions exist for the Xerox ALTO and the Intel 8080.

The design rationale for L☆ is discussed in the article, "The Kernel Approach to Building Software Systems," which appears in the 1970 Computer Science Research Review. This guide makes brief references to the principles set forth in that article. Evidence for the ease of programming with L☆ is discussed in the article "On Doing Software Experiments", which appears in the 1973 Computer Science Research Review.

L☆ is intended to be a complete system for running and constructing software systems. Completeness implies that one should be able to perform, and to construct systems for performing, the following:

A) Processing of arbitrary data types, e.g., symbolic structures, lists, numbers, arrays, bit strings, tables, text
B) Editing
C) Compiling and assembling
D) Language interpreting
E) Debugging
F) Operating system functions, e.g., resource allocation, space and time accounting, exotic control (parallel and supervisory control)
G) Communication between user and system, e.g., external languages, dynamic syntax, displays

L☆ is a kernel system. It starts with a small kernel of code and data and is grown from within the system. Thus, L☆ does not perform all the functions above when it exists only as a kernel. It does have means to construct systems for them all. Also, it does have facilities for most of these functions in the version of the system that most users will use.

L☆ is designed for the professional programmer. It assumes someone sophisticated in systems programming who wants to build up his own system and who will modify any presented system to his own requirements and prejudices. Thus, L* is intended to be transparent. All mechanisms in the total system are open for understanding and modification. No mechanisms are under the floor.

One of the design goals of the L* system was that it should be entirely self-documenting on-line to the machine, but this goal was not fully realized. There is a master source file (M-file) for L*(I) on the [A110LI00] disk area which may be used as documentation. It contains all the code for the basic system, including the kernel, bootstrap, editor, debugging tools, macro assembler, compiler, and performance monitoring tools. Also on [A110LI00], you will find an introduction and guide to the M-file (LSI.DOC) and several index files: INDEX.LSI has pointers to all source files, FINDEX.LSI is an index into the M-file by facility with a one-line description for each named symbol in the system, and AINDEX.LSI is an alphabetic index into the M-file. Also, SCPTXT.LSI is a script which demonstrates how to do some simple things in L*. It also suggests some exercises for the novice and provides help when the exercises prove to be too difficult.

Getting into L*(I) is very simple. In monitor mode, type the following:

```
.R LSIA
HELP
```

The response of the HELP function should be sufficient to get you started in the system.

The documentation for L*C.(D) is all on the [A110LC00] disk area. There is currently no M-file for this version of L*. The file LSCDOT.DOC contains much useful information about the differences between L*(I) and L*C.(D), how to use L*C.(D), and where to find further information. There are index files for the following: INDEX.LSC has pointers to all source files, FINDEX.LSC is a facility index into those source files with one-line descriptions, and AINDEX.LSC is an alphabetic index into those source files. This system is nearly compatible with L*(I), so there is currently no script for L*C.(D). To get into L*C.(D), you should follow the suggestions in LSCDOT.DOC[A110LC00].

All L* systems are nearly program compatible; a program written for one will only require minor modification to run on any other.

MACRO-10

C. D. Councill & D. Bajzek

MACRO-10 is the symbolic assembly language for the PDP-10 machine instruction set. It is characteristic of most machine languages in that it is most useful in utilizing the facilities of a PDP-10 (TOPS-10 operating system).

The PDP-10 Assembly Language Handbook (3rd ed.) is a complete reference guide for the MACRO-10 assembler since no special CMU features have been added to this language. Sections 1 and 2 contain a complete description of the PDP-10 instruction set and the MACRO-10 assembler. Section 3 contains detailed information on communication with the DEC (Digital Equipment Corporation) subset of our monitor. Chapter 4 of this section describes all of the Input/Output operators. In particular, this chapter describes the use of directory devices, disk and DECtape, which are most commonly used since they provide random access storage. Also included are diagrams and explanations of data structures and programming examples.

Unfortunately, DEC no longer publishes this gem of literature. However, CMU does possess some copies which are available for long or short term loan to department members. If you are interested in going in depth with this particular phase of the PDP-10 and wish to borrow a copy, please see or send mail to Operations. DEC now publishes a separate MACRO-10 manual which may be purchased at the PITT Bookstore. This manual doesn't contain many of the more complicated items that the Assembly Language Handbook did, but it provides a reasonable enough language manual. There is also a book written by Myron Edward White called, "Meet MACRO-10: An Introduction". This is also sold at the PITT Bookstore and provides a very good primer for the beginning MACRO hacker. Sadly, CMU, at present, doesn't have much helpful introductory on-line documentation for MACRO-10.

Below are a few concepts (operators) which are essential to creating or updating a file on disk or DECtape with a MACRO-10 program:

OPEN INIT	The OPEN and INIT programmed operators initialize a file by specifying a device (or data channel), logical device name, initial file status, and the location of the input and output buffer headers.
INBUF OUTBUF	can be used to establish buffer data storage areas.
LOOKUP	selects a file for input on the specified channel.

ENTER selects a file for output to a specified channel.

RENAME is used to:
a) alter the filename, filename extension, and the protection
or
b) delete a file associated with a specified channel on a directory device.

IN
INPUT transmits data from the file selected on the specified channel to the user's core area.

OUT
OUTPUT transmits the data from the user's core area to the file selectedon the specified channel.

CLOSE terminates data transmission on the specified channel.

RELEASE releases the data channel.

Below is a MACRO-10 program which will figure out the date from the value of the date that is stored in the operating system and will print it out on your TTY:

```
        TITLE DATE
        AC=5                        ;INITIALIZES AC TO FIVE
DATE:   SETZ    AC                  ;CLEARS (AC)
        CALLI   AC,14               ;PUTS THE MACHINE'S IDEA OF THE DATE INTO AC
        IDIVI   AC,564              ;INTEGER DIVIDE (AC) BY 564
                                    ;(AC+1) IS THE REMAINDER
        ADDI    AC,3654             ;ADD 3654 TO (AC)
        MOVEM   AC,YEAR             ;NOW (YEAR) IS THE YEAR IN OCTAL
        MOVE    AC,AC+1             ;NOW (AC) CONTAINS THE REMAINDER
        IDIVI   AC,37               ;INTEGER DIVIDE (AC) BY 37
                                    ;AGAIN (AC+1) IS THE REMAINDER
        ADDI    AC,1                ;ADD 1 TO (AC)
        ADDI    AC+1,1              ;ADD 1 TO (AC+1)
        MOVEM   AC,MONTH            ;NOW (MONTH) IS THE MONTH IN OCTAL
        MOVEM   AC+1,DAY            ;NOW (DAY) IS THE DAY IN OCTAL
        SETZ    AC                  ;CLEARS (AC)
        MOVE    AC,MONTH            ;MOVES VALUE OF "MONTH" INTO AC
        JSR     OUTNUM              ;GO TO SUBROUTINE "OUTNUM"
        OUTCHR  HYPHEN              ;PRINT A "-"
        MOVE    AC,DAY              ;MOVES VALUE OF "DAY" INTO AC
        JSR     OUTNUM
        OUTCHR  HYPHEN
        MOVE    AC,YEAR             ;MOVES VALUE OF "YEAR" INTO AC
        JSR     OUTNUM
        EXIT
        YEAR:   0                   ;INITIALIZES THE VALUES OF THE VARIABLES
        MONTH:  0                   ;TO THE SPECIFIED VALUE
        DAY:    0
        FLAG:   0
        HYPHEN: 55
        OUTNUM: 0
        IDIVI   AC,<DEC 1000>     ;INTEGER DIVIDES (AC) BY DECIMAL 1000.
        JUMPE   AC,NEXT1          ;IF AC=0, GO TO "NEXT1"
        AOJ     FLAG              ;ADD ONE TO (FLAG) AND DON'T JUMP
        ADDI    AC,60             ;ADD 60 TO (AC)
        OUTCHR  AC                ;PRINT (AC)
NEXT1:  MOVE    AC,AC+1           ;MOVE (AC+1) INTO (AC)
        IDIVI   AC,<DEC 100>      ;INTEGER DIVIDE (AC) BY DECIMAL 100.
        SKIPN   FLAG              ;SKIP NEXT INST. IF FLAG NOT EQUAL TO 0
        JUMPE   AC,NEXT2          ;IF (AC)=0, GO TO "NEXT2"
        ADDI    AC,60             ;ADD 60 TO (AC)
        OUTCHR  AC                ;PRINT (AC)
NEXT2:  MOVE    AC,AC+1           ;MOVE (AC+1) INTO AC
        IDIVI   AC,<DEC 10>       ;INTEGER DIVIDE (AC) BY DECIMAL 10.
        SKIPN   FLAG
        JUMPE   AC,NEXT3          ;IF (AC)=0, GO TO "NEXT3"
        ADDI    AC,60
        OUTCHR  AC
NEXT3:  ADDI    AC+1,60           ;ADD 60 TO (AC+1)
        OUTCHR  AC+1
        JRST    @OUTNUM           ;RETURN TO CALLING PLACE WHOSE ADDRESS
                                  ;IS STORED IN OUTNUM
        END     DATE
```

The following MACRO-10 program is a bit more illustrative. It deals with the more complicated matter of reading in a file from a disk and producing error messages should all not go smoothly. The program merely reads a string of one-digit octal numbers, ignoring all other characters, from an ASCII file called DATA.FIL. It then sums these digits and prints out their octal sum on the TTY:.

```
        TITLE ADDER
;GIVE ACCUMULATORS SYMBOLIC NAMES
        A=1
        A1=2
        DIGIT=3
        SUM=4
        COUNT=5
        PNT=5

;DEFINE I/O CHANNEL
        INCHN=1

;NOW BEGIN
START:  INIT    INCHN,1           ;INITIALIZE INPUT CHANNEL IN
                                  ;ASCII LINE MODE
        SIXBIT  /DSK/             ;LOGICAL DEVICE NAME IS DSK
        XWD     0,IBUF            ;NEED TO GIVE NAME OF INPUT
```

```
                                      ;BUFFER HEADER ONLY, SINCE WE
                                      ;ONLY WISH TO INPUT FROM THIS
                                      ;DEVICE
        JRST     NOTAVL               ;GO TO ERROR ROUTINE IF DEVICE
                                      ;IS NOT AVAILABLE
        INBUF    INCHN,1              ;SMALL AMOUNT OF DATA, WE ONLY
                                      ;NEED 1 BUFFER IN RING
        LOOKUP   INCHN,INNAME         ;LOOKUP FILE WHICH IS DESCRIBED
                                      ;IN INNAME
        JRST     NOTFND               ;ERROR IF FILE NOT FOUND

;PREPARE TO START SUMMING
        SETZM    SUM                  ;INITIALIZE SUM TO ZERO
LOOP1:  JSR      GETCHR               ;GETCHR RETURNS WITH ASCII
                                      ;CHARACTER IN DIGIT
        CAIG     DIGIT,67             ;MAKE SURE ASCII CHAR IS REALLY
        CAIGE    DIGIT,60             ;AN OCTAL DIGIT
        JRST     LOOP1                ;IF IT'S NOT, IGNORE IT AND GO
                                      ;GET ANOTHER CHARACTER
        SUBI     DIGIT,60             ;GET ACTUAL VALUE OF ASCII OCTAL
                                      ;DIGIT
        ADDM     DIGIT,SUM            ;ADD DIGIT TO SUM
        JRST     LOOP1                ;GO GET NEXT DIGIT

INEOF:  ;WHEN THE END OF FILE IS REACHED ON THE INPUT FILE
        ;THE GETCHR SUBROUTINE WILL TRANSFER CONTROL TO HERE

;NOW THE VALUE IN SUM MUST BE CONVERTED TO AN ASCII STRING
;OF OCTAL DIGITS TO BE INPUT TO THE TTY:

        MOVE     PNT,OUTPNT           ;LOAD PNT WITH A BYTE POINTER
                                      ;INTO THE AREA THE RESULT IS TO
                                      ;IS TO BE STORED INTO
        MOVSI    COUNT,-14            ;MAXIMUM OF 12 DIGIT RESULT
                                      ; (ASSUMING NO OVERFLOW)
        MOVE     A1,SUM               ;THE OCTAL DIGITS CAN BE OBTAINED
                                      ;BY SIMPLY SHIFTING THE SUM 3 BITS
                                      ;AT A TIME INTO REGISTER 4.
LOOP2:  SETZM    A                    ;INITIALIZE 4 TO BE ZERO
        LSHC     A,3                  ;MOVE LEFT 3 BITS OF A1 INTO A
        CAMN     PNT,OUTPNT           ;IF POINTER HAS CHANGED, SKIP
                                      ;OVER TEST FOR LEADING ZERO
        JUMPE    A,LEND2              ;IF THERE IS A LEADING ZERO, JUST
                                      ;INCREMENT COUNTER BUT DON'T OUTPUT
        ADDI     A,60                 ;MAKE INTO ASCII CHARACTER
        IDPB     A,PNT                ;PUT CHAR INTO TTY OUTPUT BUFFER
LEND2:  AOBJN    COUNT,LOOP2          ;IF THERE ARE MORE DIGITS LEFT,
                                      ;GO GET THEM TOO
        MOVEI    A,0
        IDPB     A,PNT                ;STORE AN ASCII NULL AT END OF STRING
        OUTSTR   OUTMSG               ;THE SPECIAL PROGRAMMED OPERATOR
                                      ;OUTPUTS AN ASCII STRING TO A TTY
                                      ; (STRING IS TERMINATED BY A NULL)
EXITT:  CALL     [SIXBIT /EXIT/]      ;SPECIAL FUNCTION TO GRACEFULLY
                                      ;TERMINATE THE EXECUTION OF A PROGRAM
```

```
;THE FOLLOWING SUBROUTINE IS USED TO INPUT ONE ASCII CHARACTER

GETCHR:                                 ;RETURN THE ADDRESS STORED HERE
GETNXT: SOSLE   IBUF-2                  ;DECREMENT THE BYTE COUNT
        JRST    GETOK                   ;NON-ZERO RESULT MEANS MORE CHARS
                                        ;LEFT IN BUFFER
        IN      INCHN,                  ;GET NEXT BUFFER FROM MONITOR
        JRST    GETOK                   ;RETURN WHEN BUFFER IS FULL
        STATZ   INCHN,740000            ;IN DOES A SKIP RETURN IF THERE WAS
                                        ;AN ERROR ON INPUT.  THE STATUS BITS
                                        ;MUST BE TESTED TO DETERMINE WHAT KIND
        JRST    INERR                   ;OF ERROR, NOT END-OF-FILE, GO PROCESS
                                        ;THE ERROR
        JRST    INEOF                   ;END-OF-FILE, RETURN TO NEXT PHASE
                                        ;OF PROGRAM

GETOK:  ILDB    DIGIT,IBUF+1            ;GET CHARACTER FROM BUFFER
        JUMPN   DIGIT,@GETCHR           ;IF NOT NULL CHAR, RETURN TO CALLING
                                        ;PLACE WHOSE ADDRESS IS STORED IN
                                        ;GETCHR
        JRST    GETNXT                  ;IGNORE NULL AND GET NEXT CHARACTER

;NEXT COME SOME ERROR ROUTINES WHICH TYPE OUT ERROR MESSAGES TO
;EXPLAIN ERRORS RECEIVED BY THE PROGRAM

INERR:  OUTSTR  INPMSG                  ;OUTPUT MESSAGE AND
        JRST    EXITT                   ;EXIT FROM PROGRAM

INPMSG: ASCIZ   /ERROR WHILE READING INPUT FILE/

;------------------------------
NOTAVL: OUTSTR  AVLMSG
        JRST    EXITT

AVLMSG: ASCIZ   /DEVICE NOT AVAILABLE/

;------------------------------
NOTFND: OUTSTR  FILMSG
        JRST    EXITT

FILMSG: ASCIZ   /FILE WAS NOT FOUND/

;------------------------------
;NOW TO DEFINE SOME CONSTANTS AND DATA

IBUF:   BLOCK   3                       ;THIS IS THE INPUT BUFFER HEADER
INNAME: SIXBIT  /DATA/                  ;NAME OF DATA FILE
        SIXBIT  /FIL/                   ;EXTENTION OF DATA FILE
        0                               ;IF THIS IS LEFT "0" THE OWNER
                                        ;OF THE FILE IS ASSUMED TO BE THE
                                        ;USER RUNNING THE PROGRAM.  THIS
                                        ;NUMBER CAN BE OBTAINED BY RUNNING
                                        ;THE "PPN" CUSP
OUTPNT: POINT   7,OUTWRD                ;POINTER TO OUTWRD WHERE THE ASCII
```

```
                                        ;REPRESENTATION OF THE SUM OF THE
                                        ;DIGITS IS TO BE STORED
OUTMSG: ASCII   /***THE SUM OF THE DIGITS IS  /
OUTWRD: BLOCK   4


        END     START
```

Now follows an example of a terminal session in which a data file for the ADDER.MAC example program is created, the program (assumed to exist in a text file on dsk) is assembled, loaded and executed.

```
.LOG C410XX00                           ;your PPN should be typed after LOG
JOB 19 CMU10A 8.3/DEC 6.02VM TTY15
Password:                               ;type your password here; it will
1508    11-Aug-77       Thur            ;not be echoed
Mon 1435...New BLIS11 on SYS:.          ;the system greet message is here
        ...SYS:NEWS(8-9)

;To run the  ADDER program, which we assume is  in an ASCII text file
;on disk  from a  previous session,  the data  file to  be used  must
;first be created.  Here we use the CREATE command which invoke LINED.

.CREATE DATA.FIL
00100   1 2 3,4,,5,6 7 8,9,E
00200   $
*E
DATS.FIL
EXIT                                    ;Now that the data file has been created
                                        ;we can execute the ADDER program
.                                       ;We can assemble, load and execute ADDER
                                        ;in three separate steps, or we can simply
.EXECUTE ADDER.MAC                      ;use the EXECUTE command to do all three
MACRO: ADDER                            ;This statement indicates that the MACRO-10
                                        ;assembler is now assembling the text file
LOADING                                 ;The loader is now loading the relocatable
                                        ; (.REL) file produced by the assembler

LOADER 1K CORE
EXECUTION                               ;Begin execution of the program
FILE WAS NOT FOUND                      ;This message is coming from the ADDER
                                        ;program.  It says that there is no file
                                        ;called DATA.FIL.  If we look back, we
                                        ;see that there is a spelling error in
                                        ;the CREATE command.
.RENAME DATA.FIL=DATS.FIL               ;we can correct this error by using the
FILES RENAMED:                          ;RENAME command to change the name of
DATS.FIL        05                      ;data file.

.EXECUTE                                ;try executing ADDER again
LOADING                                 ;Since the relocatable file already exists,
                                        ;the assembly step has been skipped
```

```
LOADER 1K CORE
EXECUTION
***THE SUM OF THE DIGITS IS 35  ;if we look back to the data file, the sum
EXIT                            ;of the digits should be 34 (we ignore
                                ;the 8,9, and E). But notice, there is
                                ;an octal non-zero digit in the line
                                ;number - the line number was included
                                ;as part of the data string!

.R PIP                          ;we can use PIP to remove the line
*DSK:DATA.FIL/N←DSK:DATA.FIL    ;numbers from the file
*↑C                             ;the /N simply causes the file to be
.                               ;written without the line numbers

.EXECUTE                        ;Again execute the program
LOADING

LOADER 1K CORE
EXECUTION
***THE SUM OF THE DIGITS IS  35 ;the sum is now correct!
EXIT
                                ;To get off the system, use the
                                ;KJOB command K/F, but remember that
.DELETE DATA.REL                ;you should delete your .REL file as
FILES DELETED:                  ;it can easily be reconstructed at a
DATA.REL        05              ;later date and shouldn't take up disk
                                ;space at the moment.
.K/F
JOB 19, USER [C410XX00]  LOGGED OFF TTY15       1545    11-AUG-77
SAVED ALL FILES (160 BLOCKS)
```

An Introduction to ITS and MACSYMA
Mark A. Sapsford
August 5, 1977

MACSYMA is probably the best known of the current symbolic manipulation systems. It is available to anyone who has access to the ARPANET, and can be of great assistance to anyone doing symbolic algebraic calculations. MACSYMA runs on a KL-10 system at MIT. ITS (the "Incompatible Timesharing System") is the operating system under which MACSYMA runs, and is in fact the OS for all of the MIT machines.

Documentation for the use of MACSYMA consists primarily of two manuals. The first is the *MACSYMA Primer*, and the other is the *MACSYMA Reference Manual*. There should be copies of both manuals in the terminal room. It is rather difficult to get private copies of the manuals, but if you really think you will need them the best idea is to talk to your adviser about acquiring them. To use MACSYMA, log on at CMU and issue the monitor command:

```
    .tn mc/a:2000
```

You will then be connected to MIT-MC, and get their introductory message. If you do not already have an account there you should now create one. This is done simply by typing:

```
    :login <user-name>
```

<user-name> should be a name not currently in use. To check whether a name is in use, type:

```
    :whois <user-name>
```

If it tells you who <user-name> is, try another one. Having found a free name and having logged in, you should run the INQUIR program to tell the system who you are. Depending on the current policy at MIT, you may already have been forced into INQUIR when you logged in, but if not type:

```
    :inquir
```

Just answer the questions as indicated. Note that your "affiliation" will probably be "T" (for Tourist).

Some useful things to know about ITS. The "top-level" is an extended DDT, which often seems a little strange but can be gotten use to. Most commands are of the form:

```
    :<command-name> <optional-arguments>
```

Useful commands are:

?	prints a list of the : commands with short descriptions
find	finds a file
info	a general information utility
listf	lists a directory
print	prints a file on your terminal
whom	lists who is on, and what they are doing

The file syntax has the form: <directory>;<filename> <extension>. Note that the separators are ";" and " " . There is a special extension, ">", which specifies the latest copy of the file. A directory of special interest is .INFO. (yes, the periods are part of the directory name). It has many useful pieces of documentation in it.

Perhaps the most useful thing that you can do at monitor level is to establish a "com link" between you and another user. When you are in a com link, anything that is typed on one terminal also appears on the other. To open a com link, you want to type:

↑←c <user-name>

Note that "↑←" is a control character, and since ↑← normally breaks an ARPANET connection, you actually type:

↑q↑←c <user-name>

The ↑q (control q) quotes the ↑←. Also, the space before <user-name> is necessary. To break the com link, you actually type:

↑q↑←n

A couple of other useful control characters are ↑z and ↑g. If you are caught in something you want to get out of, try one or the other and they should eventually get you back to some place you recognize.

Now MACSYMA. You run MACSYMA by typing:

:macsym

It will initialize itself, and then print (C1) indicating that it is waiting for input. All input prompts are of the form (C<number>), and all output labels are of the form (D<number>). Occasionally the answer has sub-expressions, and their labels are (E<number>). Input format is that of algebraic expressions as accepted by most programming languages. An input line is ended by either ";" or "$", hence input can run over more than one line. If $ is the terminator, the output is supressed. The last output expression can be referred to by "%", hence the following is possible:

```
(C1) alpha : beta + gamma;

(D1)                              GAMMA + BETA

(C2) % * 2;

(D2)                              2 (GAMMA + BETA)
```

Some very useful functions are DESCRIBE(<name>) and EXAMPLE(<name>). These will print a description and examples for <name>, respectively.

Some special constants are defined by MACSYMA. They are:

%I	The square root of -1
%PI	Pi
%E	e

An eclectic list of useful functions (courtesy of Steve Saunders):

EV(expr,arg1,...,argn)

EXPAND(expr)	RATEXPAND(expr)
XTHRU(expr)	PARTFRAC(expr,var)
MULTTHRU(expr)	FACTOR(expr)
RATSIMP(expr)	RADCAN(expr)
DIFF(expr,var,degree)	INTEGRATE(expr,var)
PART(expr,n1,n2,...)	SUBSTPART(a,expr,n1,...)
SUBST(subexprnew,subexprold,expr)	RATSUBST(a,b,c)
COEFF(expr,var,power)	RATCOEFF(expr,var,power)
REALROOTS(poly,bound)	ALLROOTS(poly)
SOLVE(eqn,var)	ENTERMATRIX(rows,cols)
LIMIT(expr,var,limitpoint,direction)	PLOT(expr,var,low,high)
IDENT(size)	DETERMINANT(matrix)
COL(matrix,num)	ROW(matrix,num)
SUM(expr,index,low,high)	PRODUCT(expr,index,low,high)

MLISP

Mike Rychener

The following is from the MLISP Manual by D. C. Smith (Stanford AIM-135, October, 1970):

"Most programming languages are designed with the idea that the syntax should be structured to produce efficient code for the computer. Fortran and Algol are outstanding examples. Yet, it is apparent that HUMANS spend more time with any given program than the COMPUTER. Therefore, it has been our intention to construct a language which is as transparently clear and understandable to a HUMAN BEING as possible. Considerable effort has been spent to make the syntax concise and uncluttered. It reduces the number of parentheses required by LISP, introduces a more mnemonic and natural notation, clarifies the flow of control and permits comments. Some "meta-expressions" are added to improve the list-processing power of LISP. Strings and string manipulation features, particularly useful for input/output, are included. In addition, a substantial amount of redundancy has been built into the language, permitting the programmer to choose the most natural way of writing routines from a variety of possibilities.

"LISP is a list-processing and symbol-manipulation language created at MIT by John McCarthy and his students (McCarthy, 1965). The outstanding features of LISP are: (1) the simplest and most elegant syntax of any language in existence, (2) high-level symbol manipulation capabilities, (3) an efficient set of list-processing primitives, and (4) an easily-usable power of recursion. Furthermore, LISP automatically handles all internal storage management, freeing the user to concentrate on problem solving. This is the single most important improvement over the other major list-processing language, IPL-V. LISP has found applications in many important artificial intelligence investigations, including symbolic mathmatics, natural-language handling, theorem proving and logic.

"Unfortunately, there are several important weaknesses in LISP. Anyone who has attempted to understand a LISP program written by another programmer (or even by himself a month earlier) quickly becomes aware of several difficulties:

A. The flow of control is very difficult to follow. In fact, it is about as difficult to follow as machine language or Fortran. This makes understanding the purpose of routines (i.e. what do they do?) difficult. Since comments are not usually permitted, the programmer is unable to provide written assistance.

B. An inordinate amount of time must be spent balancing parentheses, whether in writing a LISP program or trying to understand one. It is frequently difficult to determine where one expression ends and another begins. Formatting utility routines ("pretty-print") help; but every LISP programmer knows the dubious pleasure of laboriously matching left and right parentheses in a function, when all he knows is that one is missing somewhere!!

C. The notation of LISP (prefix notation for functions, parentheses around all functions and arguments, etc.), while uniform from a logician's point of view, is far from the most natural or mnemonic for a language. This clumsy notation also makes it difficult to understand LISP programs. Since MLISP programs are translated into LISP s-expressions, all of the elegance of LISP is preserved at the translated level; but the unpleasant aspects at the surface level are eliminated.

D. There are important omissions in the list-processing capabilities of LISP. These are somewhat remedied by the MLISP "meta-expressions", expressions which have no direct LISP correspondence but instead are translated into sequences of LISP instructions. The MLISP meta-expressions are the FOR expression, WHILE expression, UNTIL expression, index expression, assignment expression, and vector operations. The particular deficiency each of these attempts to overcome is discussed in the subsection of SECTION 3 describing the meta-expression in detail.

"MLISP was written at Stanford University by Horace Enea for the IBM 360/67 (Enea, 1968). The present author has implemented MLISP on the PDP-10 time-shared computer. He has rewritten the translator, expanded and simplified the syntax, and improved the run-time routines. All of the changes and additions are intended either to make the language more readable and understandable or to make it more powerful.

"MLISP programs are first translated into LISP programs, and then these are passed to the LISP interpreter or compiler. As its name implies, MLISP is a "meta-LISP" language; MLISP programs may be viewed as a superstructure over the underlying LISP processor. All of the underlying LISP functions are available to MLISP programs, in addition to several powerful MLISP run-time routines. The purpose of having such a superstructure is to improve the readability and writeability of LISP, long (in)famous for its obscurity. Since LISP is one of the most elegant and powerful symbol-manipulation languages (but not one of the most readable), it seems appropriate to try to facilitate the use of it."

CMU MLISP

For the user of MLISP, some knowledge of LISP is essential, especially to understand the general programming-language features of the system (e.g. dynamic scopes of variables and list processing concepts). For this manual, no script is given, relying on the introduction to LISP to give the general flavor of the system. Some examples of MLISP expressions are given below. The overwhelming advantage of MLISP is the ease of coding and understanding programs. A disadvantage is that in the actual implementation, one must deal with LISP when debugging and running a program. But the knowledge of LISP syntax need only be a "reading" knowledge - the LISP editor helps alleviate many of the difficulties of LISP syntax (especially parenthesis counting). The mapping of MLISP code to LISP is sufficiently straightforward that dealing with two representations of a program is not troublesome.

Typically an MLISP user creates a program using SOS or some other text editor, runs MLISP to translate the program to LISP, and then runs and debugs in LISP (MLISP itself is a LISP program residing in a LISP core image along with the editor and debug packages). Corrections made in the LISP core image during debugging must later be made to the SOS version also, which is probably the only deficit in time spent. This extra editing time is easily offset by the faster coding time and the likelihood of fewer syntax errors in code. MLISP is so efficient that little time difference can be noticed in reading in an MLISP file versus reading in the LISP equivalent. MLISP also provides syntax checking and error messages, while LISP does so only rarely.

There is a file on the system, MLISP.DOC, which describes quirks and manual updates for the local version of MLISP. There is a program, MEXPR, for converting LISP to MLISP. There is a function in MLISP, MTRANS, which allows the user to evoke the MLISP translator to convert expressions read in under program control. There is also MEVAL, a top-level READ-MTRANS-EVAL-PRINT loop so that interaction with MLISP can take place in MLISP expressions rather than in LISP.

MLISP QUICK REFERENCE

This short document is intended to:

a. Give some examples of the advantages of MLISP syntax over LISP;

b. Provide a quick reference to users of MLISP who have some exposure to the MLISP manual; it will not be sufficient in itself to introduce MLISP except to the very experienced LISP user who is also willing to guess and experiment.

LISP -> MLISP:

```
(APPEND A B)  ->  A @ B
(COND  (C1 E11 E12 E13)
       (C2 E21 E22 E23 E24)
       (CN EN1))
  ->
       IF C1 THEN E11 ALSO E12 ALSO E13
         ELSE IF C2 THEN E21 ALSO E22 ALSO E23 ALSO E24
         ELSE IF CN THEN EN1;

(DE FCN (X) EXPRES)  ->  EXPR FCN(X); EXPRES;
(DF FCN (X) EXPRES)  ->  FEXPR FCN(X); EXPRES; % SIMILARLY FOR LEXPR, MACRO %
(LAMBDA (V1 V2 VK) EXPRES) ->  LAMBDA(V1,V2,VK); EXPRES
((LAMBDA (X Y Z) EXPRES) 1 2 3)  ->  LAMBDA(X,Y,Z); EXPRES; (1, 2, 3)
(PROG (V1 V2 VN) E1 E2 E3)  ->  BEGIN NEW V1,V2,VN; E1; E2; E3 END

(LIST A B C)  ->  <A, B, C>
(OP1 ARG1)  ->  OP1 ARG1
            ->  OP1(ARG1)
(OP2 ARG1 ARG2)  ->  ARG1 OP2 ARG2
                 ->  OP2(ARG1, ARG2)
(OPN ARG1 ARG2 ARGN)  ->  OPN(ARG1, ARG2, ARGN)
(QUOTE (A B C))  ->  '(A B C)
(SETQ X E1)  ->  X ← E1
/.  ->  ?.
*  ->  ?*
```

```
~ Comment
  ->  % Comment %
```

Precedence Examples: MLISP -> LISP

```
A CONS B CONS NCONS C  ->  (CONS A (CONS B (NCONS C)))
A CONS B @ C CONS D  ->  (CONS A (APPEND B (CONS C D)))
CAR L + B  ->  (PLUS (CAR L) B)
CAR A ← L  ->  (CAR (SETQ A L))
A + B * C = D / E OR A EQ C AND B LESSP D
   ->  (OR (EQUAL (PLUS A (TIMES B C)) (QUOTIENT D E))
          (AND (EQUAL A C) (LESSP B D)))
A + B CONS C / D  ->  (CONS (PLUS A B) (QUOTIENT C D))
X ← L @ M  ->  (SETQ X (APPEND L M))
X SET L @ M  ->  (APPEND (SET X L) M)
NOT A = 0  ->  (EQUAL (NOT A) 0)
NOT (A = 0)  ->  (NOT (EQUAL A 0))
RETURN CAR L CONS X  ->  (CONS (RETURN (CAR L)) X)
RETURN(CAR L CONS X)  ->  (RETURN (CONS (CAR L) X))
RETURN A ← B  ->  (RETURN (SETQ A B))
C ← A EQ B  ->  (SETQ C (EQ A B))
(B ← C) EQ D  ->  (EQ (SETQ B C) D)
A + B ← C + D  ->  (PLUS A (SETQ B (PLUS C D)))
```

Examples of commonly-used looping statements:

```
FOR NEW X IN L DO BODY;  % loop on top-level elements of list L %
FOR NEW X ON L DO BODY ; % loop on cdr's of list L %
FOR NEW X ← 1 TO 10 BY 2 DO BODY;
FOR NEW Y ← 1 TO 11 DO BODY UNTIL CONDITION;
FOR NEW X IN L COLLECT BODY; % APPENDs together values of body, each iteration %
FOR NEW I IN L; FUNC BODY; % applies FUNC to body each time, cumulatively %
FOR NEW X IN L FOR NEW Y ← 1 TO 9 DO BODY; % ranges in parallel %
DO BODY UNTIL CONDITION;
WHILE CONDITION DO BODY;
```

PASCAL

Andy Hisgen

Pascal is an Algol-like language. Its main extensions relative to Algol 60 lie in its data structuring facilities. Pascal is a typed language. All variables have a data type, which essentially defines the possible values which may be assumed by that variable. There are four standard basic types: boolean, integer, real, and char. The user may define new types in several ways: by enumeration (that is, by listing the possible symbolic values), by specifing a subrange of another scalar type, and by defining a structured type or a pointer to a structured type. A structured type is defined by describing the types of the components and the structuring method: array, set, record, or file.

References (some of these may be borrowed from Andy Hisgen)

[1] N. Wirth
The Programming Language Pascal
Acta Informatica Vol. 1, pp.35-63, 1971.

[2] K. Jensen, N. Wirth
Pascal - User Manual and Report
Lecture Notes in Computer Science Vol. 18
Springer Verlag Berlin, Heidelburg, New York 1974.
(Several copies are in Engineering and Science library).

[3] N. Wirth
Systematic Programming
Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[4] N. Wirth
An Assessment of the Programming Language Pascal.
Proceedings of the International Conference on Reliable Software,
April 21-23, 1975, pp.23-30.

[5] C.A.R. Hoare and N. Wirth
An Axiomatic Definition of the Programming Language Pascal.
Acta Informatica Vol. 2, pp.335-355, 1973.

[6] C.A.R. Hoare
"Notes on Data Structuring", in Structured Programming, by
O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Academic Press,
London and New York, 1972.

[7] A.N. Habermann
Critical Comments on the Programming Language Pascal.
Acta Informatica Vol. 3, pp.47-58, 1973.

[8] DOC:PASCAL.DOC , an online file on the PDP-10A. Includes information
on the PDP-10 implementation of Pascal.

AN EXAMPLE

```
!Explanatory comments are preceded by a "!".

!We have a file TOY.PAS which contains a Pascal program.
!We TYPE it on the terminal.

.TYPE TOY.PAS
(*this is a Pascal comment, between left paren asterisk and
asterisk right paren*)
(*toy example*)
type
  ToyKind = (Ball,Top,Boat,Doll,Blocks,Game,
            Model,Book,OtherToy);
               (*ToyKind is an "enumeration type".  It provides us
               some symbolic constants*)
  Bentham = (Alot,Some,Alittle,None);
  PriceRange = 0..1000;
               (*PriceRange is a "subrange type".  It is a subrange
               of the integers.*)
  PToy = ↑Toy;
               (*The ↑ means "pointer to".  Thus, the type PToy is
               a pointer to Toy.*)
  Toy  = record
           Kind : ToyKind;
           Cost : PriceRange;
           Enjoyed : Bentham;
           Broken, Lost : boolean;
           NextToy : PToy
         end; (*Toy record declaration*)

  ToyBox = PToy;                 (*a toybox is represented as
                                 a list of toys*)

var
  MyToyBox : ToyBox;

procedure AddToToyBox( AddToy:PToy; var B:ToyBox );
begin
AddToy↑.NextToy := B;  (*the ↑ does a dereference of the pointer to
                       toy yielding a toy.*)
B := AddToy
end; (*AddToToyBox*)

function MakeToy(K:ToyKind; C:PriceRange):Ptoy;
var P:Ptoy;
begin
new(P);          (*allocates a new record of type toy and makes P
                 point at it.*)
with P↑ do begin
```

```
  Kind := K;  Cost := C;
  Enjoyed := None;  Broken := false;  Lost := false;
  NextToy := nil;
  end;
MakeToy := P
end; (*MakeToy*)

function ToyAssets( B:ToyBox):PriceRange;
var P:ToyBox;
 S:PriceRange;
begin
S := 0;
P := B;
while P<>nil do begin
  S := S + P↑.Cost;
  P := P↑.NextToy;
  end;
ToyAssets := S;
end; (*toyassets*)

procedure xmas;
begin
AddToToyBox( MakeToy(Boat,399), MyToyBox);
AddToToyBox( MakeToy(Top,499), MyToyBox);
end; (*xmas*)

begin (*main program*)
MyToyBox := nil;
xmas;
writeln(tty,'After Xmas toy assets are :', ToyAssets(MyToyBox));
end.

!We run the pascal compiler.

.R PASCAL
Enter command:  RELfile,LSTfile←PAScal file  !It prompts us.
*TOY            !Responding with just the name of the program will
                !generate a .REL file and omit the listing file.

PASCAL VERSION OF 14-Jul-77
File: TOY   .PAS

NO ERROR DETECTED

Highseg:   1K
Lowseg :   1K
Runtime:   0: 0.266

EXIT
```

```
.EX TOY           !We link and execute the program by using
                  ! the monitor EXECUTE command.
LINK:  Loading
[LNKXCT TOY Execution]
After Xmas toy assets are :          898

EXIT
```

SAIL

INTRODUCTION

SAIL is a high-level programming language for the PDP-10 computer. It includes an extended ALGOL 60 compiler and a companion set of execution-time routines. In addition to ALGOL, the language features: (1) flexible linking to hand-coded machine language algorithms, (2) complete access to the PDP-10 I/O facilities, (3) a complete system of compile-time arithemtic and logic as well a s a flexible macro system, (4) user modifiable error handling, (5) backtracking, and (6) interrupt facilities. Furthermore, a subset of the SAIL language, called LEAP, provides facilities for (1) sets and lists, (2) an associative data structure, (3) independent processes, and (4) procedure variables. The LEAP subset of SAIL is an extension of the LEAP language, which was designed by J. Feldman and P. Rovner, and implemented on Lincoln Laboratory's TX-2 (see [Feldman and Rovner]). The extensions to LEAP are partially described in "Recent Developments in Sail" (see [Feldman]).

SAIL was developed at the Stanford Artificial Intelligence Laboratory and is currently running on at least a dozen PDP-10 installations. At CMU, it is used extensively by the speech, vision, and graphics efforts, among others.

The SAIL compiler can be invoked in the same ways as FORTRAN or MACRO. The Default extension for SAIL SOURCE PROGRAMS is .SAI.

The COMPILE, EXECUTE, LOAD, or DEBUG commands may be used. For example:

```
.EXECUTE PRGRAM.SAI
.DEBUG PRGRAM              (where the extension is the default for
                             SAIL, .SAI)
.EXECUTE PROG1, SUB1, SUB2  (where SUB1 and SUB2 are separately
                                       compiled procedures)
```

For details on these commands, see the PDP-10 Command Manual.

If you use DEBUG, EXECUTE, LOAD, etc., they will do the above things correctly automatically upon seeing the .SAI extension.

REFERENCES

[1] SAIL User Manual, CMU version of June, 1973, available from Computer Science Department. This manual describes the SAIL language and the execution-time routines for the typical SAIL user: a non-novice programmer with some knowledge of ALGOL. It lies somewhere between being a tutorial and a reference manual.

[2] Most recent CMU manual update, available from Computer Science Department.

[3] Erman, L., SAIL Pocket Guide (Sailing Chart), available from Computer Science Department. One section of this could be entitled: "How to Start Using SAIL if you already Know Algol."

[4] Feldman, J. and F. Rovner, "An Algol-Based Associative Language," CACM, 12(8), August, 1969, pp. 439-449.

[5] Feldman, J. A., Low, J. R., Swinehart, D. C., and Taylor, R. H., Recent Developments in Sail. AFIPS Fall Joint Conference, 1972, 1193-1201.

[6] Two video tapes (T148, T149): "Introduction to SAIL for Those who Know Algol."

EXERCISES

1. Write a SAIL program to merge two SOS files, according to sequence numbers.

2. You are given an M x N matrix of numbers where M and N can be very large. The values of the entries are 0 - 15. In order to conserve DISK space, it is desirable to pack the data (each number can be represented in 4 bits) nine entries to a PDP-10 word before writing the matrix onto a DISK file. Write a SAIL program which does this packing, writes out the file, reads it in, and "unpacks" it.

SOME SIMPLE PROGRAMMING EXAMPLES

```
[1] BEGIN "FACTORIAL"
    COMMENT THIS PROGRAM READS NUMBERS FROM THE TELETYPE AND
            TYPES BACK THEIR FACTORIALS;
    REQUIRE "BAYSAI.SAI[A710SA00]" SOURCE!FILE;
    COMMENT THIS IS A CONVENIENT PACKAGE;

    INTEGER PROCEDURE FACT(INTEGER N);
         BEGIN "FACT"
         INTEGER I;
         I←1;                   !  INITIAL VALUE FNF THE LOOP;
         FOR N←N STEP -1 UNTIL 1 DO
         I←I*N;                 !  NOTE THAT FOR N=0, I WILL BE 1;
         RETURN(I);
         END  "FACT";

    INTEGER X;
    WHILE TRUE DO
         BEGIN "INFINITE LOOP"
         ! WHEN FINISHED WITH THE PROGRAM, TYPE C TO BREAK OUT;
         OUTSTR(CRLF&"NUMBER, PLEASE:");
         X←CVD(INCHWL);         !  READ THE NUMBER;
         OUTSTR(IF X<0 THEN "NOW REALLY" ELSE CVS(FACT(X)));
         END  "INFINITE LOOP";

    END "FACTORIAL";
```

SITBOL

J. Dills, S. Schlesinger, M. Shaw

SITBOL is a PDP-10 dialect of SNOBOL4, the string-processing language developed by Bell Laboratories. It is a full SNOBOL4, but substantially faster than the version supplied by DEC and better tailored to the PDP-10 environment.

The language has operations for joining and separating strings, testing their contents, and making replacements within them. Strings can be broken down and reassembled differently. The language also provides the ability to describe a set of character patterns and search a given string to match a substring to a pattern. Because SNOBOL4 is mainly character oriented, the numerical capabilities with both integers and reals exist, but are limited. Arrays and tables are available.

Execution of SNOBOL4 is interpretive. This allows easy tracing of variable values, and the ability to redefine functions during execution. The language can be extended by using data type definition facilities and defining operations on these through function definition (i.e., lists, complex numbers).

The basic reference for SITBOL is the SNOBOL4 text [1]. The changes to I/O and the additional features are described in SITBOL.DOC [2].

REFERENCES

[1] Griswold, R. E., J. F. Poage, and I. P. Polonsky, The SNOBOL4 Programming Language, Prentice Hall, 1971.

[2] SYS:SITBOL.DOC, a printable text file on the PDP-10.

RUNNING SITBOL PROGRAMS

Enter the SITBOL system by typing 'R␣SITBOL' while in command mode. When it responds with '*', you may type command strings. The command string takes the form

```
*OUTPUT-FILE, LISTING-FILE = INPUT-FILES
```

(blanks surrounding delimiters are optional) where OUTPUT-FILE = the file which receives strings assigned to the variable output. If the OUTPUT FILE is absent, TTY: is assumed. If the extension is absent (but the name present), .LST is assumed.

LISTING-FILE = the file containing the source listing and statistics. If absent, no listing is generated. If only the extension is absent, .LST is assumed.

INPUT-FILES = a sequence of input files separated by commas (a null file implies TTY:, ↑Z is the end-of-file) containing source and input data. A reference to input will begin reading from the line following the end label. No end-of-file indication is given at file boundaries.

For bells and whistles, see SITBOL.DOC [2].

Beware! Keywords and other alphabetics which are part of the language must be in upper case. This includes the F and S that appear in gotos.

SITBOL I/O NOTES

SITBOL is simpler than SNOBOL4 I/O as described in Griswold [1]. To perform input and output from a SITBOL program, variables are associated with devices or file names. If a variable is associated in an output relation with a device or file, then each time the variable is assigned a value, a copy of the value is written to the device or file. Similarly each time an input variable is used, a new value is read from the associated device or file to become the value of the variable.

Input and output associations have the form

```
INPUT(<name>,<stream>,<format>)
OUTPUT(<name>,<file>,<format>)
```

where

<name> is a string containing a variable name

<file> is a string containing a file specification in the usual format with normal defaults:

```
<device>:<name>.<ext> [<PPN>] <priv>
```

Note that CMU PPNs do not work and you will usually ignore <priv>.

<stream> is a string containing a series of file specifications separated by commas

<format> is null for normal line-by-line I/O. See SITBOL.DOC for other options.

Thus the commands

```
INPUT('SOURCE','DATA.RAN',)
OUTPUT('SINK','RESULT.NEW',)
```

make SOURCE an input variable (new value from file DATA.RAN for each fetch) and SINK an output variable (sends line of output to file RESULT.NEW for each store).

The variables INPUT, OUTPUT, and TTY are predefined with associations to the obvious things (TTY for both input and output).

SAMPLE PROBLEMS

Write SNOBOL programs to do the following:

1. Read and print cards, removing all blanks before printing.
2. Read cards and print those beginning with '/'.
3. Read cards and print those not containing '*'.
4. Reverse the order of characters in a string.
5. Count all the vowels in the input text.
6. Read left-justified text; print it centered on the line.
7. Alphabetize the characters of a string.
8. Count the occurrences of pronouns in English text.
9. Read a deck. For each card, if a vowel appears in the first five columns, print the card as it was read. If not, and if '$' or '*' appears between columns 60 and 70, reverse the card, prefix two slashes, and print the result.
10. Read numbers in free form (e.g., separated by commas). Every time you have read ten numbers, print them in columnar format. Assume that no number is more than ten characters long.
11. Devise a simple cipher (e.g., letter substitution). Write programs to encode and decode messages using this cipher. Generalize to accept a description of the cipher as an input. How complex can you make the cipher?

SITBOL Script

This program reads strings from a file named REV.DAT, reverses them, and prints the original and reversed strings in the default output file.

```
.CREATE REV.SNO
00100             DEFINE('REVERSE(X)A)                              :(REVEND)
00200   REVERSE   X  LEN(1) . A = ''                                :F(RETURN)
00300             REVERSE = A REVERSE                               :(REVERSE)
00400   REVEND
00500             INPUT('SOURCE','REV.DAT')
00600   RVLOOP    DATA = TRIM(SOURCE)                               :F(END)
00700             OUTPUT = DATA ' REVERSED IS ' REVERSE(DATA)   :(RVLOOP)
00800   END
*E
REV.SNO
EXIT
.
.
.CREATE REV.DAT
00100   ABCDEFGHIJKL
00200   1234567890
00300   $
*E
REV.DAT
EXIT
.
.R SITBOL
*REV
        00100           DEFINE('REVERSE(X)A)                        :(REVEND
.)

'
*****UNMATCHED QUOTE*****

? ERROR 4.2 IN STMT 100 AT LEVEL 0
DETECTED BY COMPILER
*↑C
.
.
.EDIT REV.SNO
REV.SNO
*R100
00100           DEFIND\D\E('REVERSE(X)A')                 :(REVEND)
*P100
00100           DEFINE('REVERSE(X)A')           :(REVEND)
*E
REV.SNO
EXIT
.
.
.R SITBOL
*REV
ABCDEFGHIJKL REVERSED IS LKJIHGFEDCBA
1234567890 REVERSED IS 0987654321
*
*REV,REV=REV
*↑C
```

Create the input file

Short form of command line defaults OUTPUT to the terminal

SOS line numbers appear with error messages (if they are in the source file)

Control-C to leave SITBOL

Correct the error found earlier

Correct execution

Full form of command line creates LST fil and changes OUTPUT default to the LST fil which can then be typed or printed.

```
.TY REV.LST
SITBOL (VERSION 4A  -  JUNE, 1974)
STEVENS INSTITUTE OF TECHNOLOGY - HOBOKEN, N. J.




00100              DEFINE('REVERSE(X)A')             :(REVEND)
00200   REVERSE  X  LEN(1) . A = ''                          :F(RETURN)
00300              REVERSE = A REVERSE                       :(REVERSE)
00400   REVEND
00500              INPUT('SOURCE','REV.DAT')
00600   RVLOOP   DATA = TRIM(SOURCE)                         :F(END)
00700              OUTPUT = DATA ' REVERSED IS ' REVERSE(DATA)   :(RVLOOP)
00800   END
ABCDEFGHIJKL REVERSED IS LKJIHGFEDCBA
1234567890 REVERSED IS 0987654321




 NORMAL TERMINATION AT LEVEL 0 IN STMT 600

  SITBOL STATISTICS SUMMARY
     200 MS. COMPILATION TIME
     134 MS. EXECUTION TIME
    2481 MICROSECOND AVE. PER STMT EXECUTED
      54 STMTS EXECUTED,        3 FAILED
       1 GARBAGE COLLECTIONS
       2 READS
       2 WRITES
     11K HIGH SEGMENT
      4K MAX LOW-SEG
      4K CURRENT LOW SEGMENT COMPRISING:
         2059 WORDS FLOATING
         1116 WORDS GROWING
          549 WORDS FOR STACKS:
                82 NAME-LIST STACK, MAXIMUM WAS 4
               137 PATTERN MATCHING STACK, MAXIMUM WAS 7
               330 SYSTEM STACK, MAXIMUM WAS 29
           37 WORD VARIABLE TABLE
```

1. Introduction

This chapter briefly describes the set of programs you can use to produce various sorts of documents, such as technical reports. A number of the facilities involved, such as the XGP, are described elsewhere.

A few document formatting terms are needed for you to understand the rest of this chapter. *Filling* is the process of taking input lines of random lengths, splitting the input into *words*, which are streams of text delimited by blanks and line terminators and other *word breaks*, and fitting as many such words onto an output line as will fit properly. For example, if we take the input text

```
This is a group of lines
of random length to
illustrate filling and justification.
```

and use it to fill 40-character output lines, we get

```
This is a group of lines of random
length to illustrate filling and
justification.
```

Justification is the process of inserting extra spaces in a line so that the right margins, as well as the left margins, are aligned. When the previous example is filled and then justified, it becomes

```
This  is  a group  of  lines  of  random
length   to   illustrate   filling   and
justification.
```

In addition to being filled and justified, lines may be *centered*:

```
       This is a group of lines
         of random length to
 illustrate filling and justification.
```

or *flushed right*:

```
             This is a group of lines
                  of random length to
 illustrate filling and justification.
```

There are several ways to accomplish this and related tasks; the following sections describe some of them briefly.

1.1. Files and Documents

In formatting written text, the ultimate goal is to bring it to the eyes of the reader. It may be printed on paper, displayed on a CRT, filmed on microfiche for later use in a fiche reader, or whatever. The *production* of a document is the conversion of a source file on the computer into one of those forms.

For display on a CRT, a source file may be typed directly. One frequently finds, however, that the file is more readable if its lines are nicely filled and the paragraphs uniformly indented. The SOS editor may be used to manually reformat a file for this purpose. If the file is long, or is to be frequently changed, or contains many special effects, it is probably worthwhile to use one of the *document formatting* programs to translate a source file into a formatted output file. Using one of these programs, the source file is "compiled" into an output file, which is then suitable for display on a CRT.

For paper output, either the line printer or the XGP is used. Files for the line printer differ from files for display on the CRT only in that they have page-eject codes in them periodically. The XGP, the Xerox Graphics Printer (on which this chapter and most others in this report were printed), is used for most paper document production at CMU. The XGP may be used with variable-width character sets, requiring special-format output files to be produced. XOFF and PUB are document compilers that produce XGO files from document source files; these XGO files can then be printed on the XGP to produce paper output. The XGP may also be used as a simulated line printer to print source files, using fixed-width character sets, but the output is not as satisfactory as that produced with variable-width character sets.

1.2. Simple Formatting with SOS

The SOS editor can do simple formatting tasks. Its formatting is controlled by three variables, called LMAR, RMAR, and PMAR. LMAR and RMAR control the left and right margins; all of the final text will be between the columns indicated by these two numbers. PMAR specifies how far the first line of a paragraph is to be indented.

JF<range> will fill all the lines in the indicated range. The first line in the range, and every line which follows a blank line, is considered to be the start of a new paragraph. The command JU<range> will fill and justify all of the lines in the indicated range. The variable JMAX limits the number of extra blanks that can be inserted to try to justify a line. JC will center, and JR will flush right.

1.3. XOFF

For somewhat more complicated formatting, there is a program called XOFF. It takes an input file which contains commands to XOFF imbedded in the text, and produces a formatted output file. The file SYS:XOFF.DOC describes XOFF.

XOFF currently is not maintained or supported by anyone at CMU. It is out of date, and doesn't really interface correctly with the XGP system. Volunteer maintainers will be accepted gladly.

1.4. PUB

The most commonly-used document formatting program is PUB. Like XOFF, it takes an input file with commands imbedded in the text and produces a formatted output file. The PUB command language resembles block-structured programming languages such as ALGOL and SAIL. It contains conditionals, BEGIN-END groupings, repetition, and string and integer variables and expressions. It also provides a facility for user-defined macros and procedures.

PUB distinguishes between *command lines* and *text lines*. A command line has a dot (.) as the first character of the line. The remainder of the line is assumed to contain PUB commands.

A PUB source file should have extension ".PUB". You can invoke PUB by saying one of the following:

```
COMPILE file.PUB
COMPILE file.PUB(X)
```

The first form will produce a file with extension ".DOC", suitable for listing on the printer. The second will produce a file with extension ".XGO", suitable for listing with the XGP.

PUB is badly documented; there may be a very old manual somewhere in the terminal room. Don't believe everything that you read in it. The file FEB76.DOC[A700PU00] contains all of the on-line documentation, including command summaries. The file BUGSIN.PUB[A700PU00] describes most of the known PUB bugs. There are many bugs in PUB, and most of them will never be fixed. The program is so convoluted, and has been rebuilt and modified and hacked to such a degree that it is just not worth while to invest any effort in fixing them. Most of them can be circumvented.

1.5. The SONGBOOK Macros

As noted previously, PUB is poorly documented and very difficult to learn. The next chapter describes a set of macros which simplify the process of using PUB to produce papers and technical reports. The macros also provide a starting place for learning PUB. If you are curious as to how a particular effect is achieved, you can examine the macro that does it. They are called the Songbook macros because they were originally created for the Hydra Songbook, an introductory guide to Hydra. Why this introductory guide is called the *Songbook* is one of those questions whose answer is clouded in legend and will probably never be known. This chapter was produced with these macros; the source file is SONGBK.PUB[A700PU00].

Typically, a PUB file using these macros would look like

```
.source!! (SONGBK);
.every heading(, {page},);
.setfont(TXFONTA,A); setfont(TXFONTB,B);
.
.begin "title page"
.center
My Document
H. Q. Bovik
August 5, 1977
.end "title page";
.precontents;
This is an unnumbered preface, which precedes the
table of contents.
.endprecontents;
.
.sourcefile(|first.chp|);
.sourcefile(|last.chp|);
```

Most of this will be explained in the next chapter. The first line causes PUB to read in the definitions of the SONGBOOK macros. The main body of your text can consist of lines of any length; the text will all be filled and justified. You indicate the start of a new paragraph by typing a blank line. The two SOURCEFILE statements (see 2.5.2) illustrate the common practice of keeping each major division, such as a chapter, in a separate file.

A reasonable subset of the effects described below can be achieved without including any PUB commands at all. If none of your lines start with a dot (.), if you insert a blank line in front of every paragraph, and if you never use the at-sign (@) character in your text, then you will get neatly indented, filled, and justified output. To get just the text responses described in 2.2, you need only have

```
.source!! (ATSIGN);
```

as the first line of the file.

2. The SONGBOOK Macros

2.1. Special Characters

A number of characters have special meaning to PUB; you must often be careful of them when creating your text file. The rest of this section assumes you are using the SOS editor for this task, and know a little about its escape conventions.

The character α (alpha, or ?" in SOS) causes the next character to be treated as ordinary text. It thus *quotes* the following character, disabling any special effects it might normally cause. To get one of the characters described below to appear in your text, you should quote it with α.

The character \ (back slash) is used as a tab character inside examples (see 2.5.4), tables (see 2.5.5), and itemized lists (see 2.5.3). Elsewhere it is an ordinary character. Tab stops can be set with the TABS command, which takes a list of column numbers as an argument. By default, there is always a tab stop in the last column of the page.

The character # (hash mark) is converted to a space in the output file. It is used to insert spaces without ending a "word", so that PUB will not insert any extra spaces at that point, and will not break the "word" across lines.

The characters { and } can be used to invoke PUB commands in the middle of a text line. Everything between the braces is passed to the PUB command scanner, which evaluates them. The most common use of this is to evaluate some PUB expression which should be inserted in the text. For example,

```
    This is page {page}
```

causes the contents of the PUB variable PAGE (the current page number) to be inserted in the sentence.

2.2. Text Responses

A *text response* is a sequence of characters imbedded in the middle of text lines; the sequence is usually used to achieve certain special effects. This section describes text responses for a set of common functions.

2.2.1 Special Effects

The source files define text responses used to perform special purpose functions. A text line of the form

```
    text1@u[TEXT]text2
```

will produce an underlined version of the enclosed text, namely

```
    text1TEXTtext2
```

On any one line the XGP can use up to two *character sets*, or *fonts*; they are called the A and the B fonts. The response

```
    @i[TEXT]
```

switches to the B character set to print the enclosed text. The "i" stands for "italic"; a typical use for the response is to switch to an italic character set. Because of timing problems in the XGP, you must have loaded the B character set at least one full output line before you use the new character set. There is also a @b[...] response, intended to switch to a boldface character set. Since the XGP can only handle two character sets at a time, both @b and @i at the moment do exactly the same thing, namely switching to the B character set.

The text response @+[text] accomplishes $^{\text{superscripting}}$ of the bracketed text. The text response @-[text] accomplishes $_{\text{subscripting}}$ of the bracketed text.

2.2.2 Footnotes

The text response

```
    @foot< This is
    a footnote >
```

makes a footnote out of the text between the "<" and the ">". The A and B fonts in the footnote are controlled by the variables FOOTFONTA and FOOTFONTB. The above example produces the footnote at the bottom of the page[1]. Because of an obscure and essentially unfixable bug in PUB, you may not use both A and B footnote fonts in a document that requires more than one footnote per page. If you require more than one footnote per page, then either change your writing style or set FOOTFONTB to "CLEAR".

2.3. Sectioning

These macros create section, subsection, (etc.) titles and make entries in the table of contents. All of them are of the form

```
    name(title,tag)
```

[1] This is a footnote

NAME is the name of the macro. TITLE is a character string which provides the title for the section. TAG is optional; it can be used to associate a label with the division in question. The label can be used with the YON macro to provide cross-references. If a tag is provided it must end with a colon; e.g.

```
    .chap(|The Rutabaga|,ruta:)
```

produces a chapter titled "The Rutabaga", which can be referenced elsewhere as {yon ruta}. The curly braces cause PUB to go into command mode to evaluate the macro. In this document the section on itemized lists is labelled LISTS; the text "(see {yon lists})" produces "(see 2.5.3)".

The MAJORPART macro begins a new major part, that is, a division labelled "PART" in the Songbook. Each major part begins on a new page. The part number and title are printed on the terminal when PUB begins to process the part. CHAP begins a chapter. Chapters are numbered sequentially from 1, and are not renumbered within each new major part. The first chapter of a major part begins on the same page as the part title; all other chapters begin on a new page. The chapter number and title are printed on the terminal when PUB begins to process the chapter. UNNUMBERED begins a division at the same level as a chapter, which has no associated number. It is intended for such things as a preface, forward, prologue, epilogue, or bibliography. SEC begins a section. Sections are numbered sequentially within chapters. SUBSEC begins a subsection. Subsections are numbered sequentially within sections. PARA begins a labelled paragraph. Paragraphs are numbered sequentially within subsections. The macro SUBSUBSEC is exactly equivalent to PARA. The first line of the paragraph begins on the same line as the title; for all of the other sectioning macros, the title and the first line of the section are separated by at least one blank line. APPENDIX begins an appendix. Appendices behave like chapters, except that their indices print as upper case alphabetics rather than Arabic numbers.

For each of these macros there is a corresponding one formed by prepending "eval!" to the front of the macro name; this form of the macro takes an expression as an argument, rather than a string. Thus if the PUB variable GORP contains the string "XXXX" then CHAP(GORP) produces a chapter titled GORP, while EVAL!CHAP(GORP) produces a chapter titled XXXX.

2.4. Character Sets

2.4.1 Changing Fonts

The macro

```
    .setfont(FONT,CHARSET)
```

is used to change the A or the B character set. CHARSET is the single letter A or B, either upper or lower case. FONT is a character font name, such as NGR25, or a

Songbook character set variable, such as EXFONTA. The Songbook character sets are summarized in the following table, giving the Songbook name, the corresponding actual character set, and a brief description of its purpose.

TXFONTA	NGR25	main text font
TXFONTB	NGI25	secondary text font
EXFONTA	FIX25	Main ("A") font for examples
EXFONTB	FIX25	Secondary ("B") font for examples
HOFONT	NGI25	Paragraph titles in text.
H1FONT	NGB25	Subsection titles in the text; section title numbers in the table of contents
H2FONT	NGB30	Section titles in the text; chapter titles in table of contents
H3FONT	NGR40	Chapter titles in the text
H4FONT	BDR40	Part titles in text and table of contents.
FOOTFONTA	NGR20	main footnote font
FOOTFONTB	NGR20	secondary footnote font

If you have a font you are using in several places, it is a good idea to assign the font to a variable, and use the variable name in SETFONT requests. This is exactly what was done with all of the Songbook fonts mentioned above. For example,

```
.MYFONT ← NGR40;
...
.setfont(MYFONT,B)
```

Because of timing problems in the XGP, a character set must be loaded at least one full output text line before it is used. Thus you must place your SETFONT requests a reasonable distance before you use the character set, and you cannot use more than two character sets on a line. Also, changing the A character set can cause problems; ask for help before you try.

2.4.2 Defining New Fonts

The macro

```
.DEFINE!FONT(NAME,FILE,ID!NO,HEIGHT,WIDTH,BFILE)
```

can be used to tell PUB about non-standard character sets you may wish to use. NAME is an identifier that you can later use in the SETFONT macro to refer to the character set. FILE is the name of the file containing the character set. ID!NO is a number that the XGP system and PUB will use to refer to the character set; it must be greater than 32 and less than 2^{14}. HEIGHT and WIDTH are the height and average width of the character set in terms of XGP raster points; you can find these numbers by examining the character set file with BILOS. Typically, the last two characters of the name of the character set file give the height of the character set. For example, NGR25 is 25 raster points high.

The XGP is usually attached to the PDP-10/B, while most people using PUB are on the PDP-10/A. Both PUB and the XGP must read the files describing character sets. If the file containing the character set on the /B is in a different place than on the /A, the parameter BFILE should be the name of the file on the /B. If both are in the same place, BFILE may be omitted.

2.5. Miscellaneous Macros

2.5.1 Page Headings

PUB can be told to place headings and footings on every page. The macros which cause this to happen are all of the form

```
.NAME(LEFT, CENTER, RIGHT)
```

The first argument is left-flushed, the second is centered, and the third is right-flushed, at the top of every page. If the arguments are enclosed in curly braces {}, they will be interpreted as PUB expressions and will be evaluated each time they are produced. Thus the EVERY HEADING given in the skeleton in section 1.5 causes the page number to be inserted in the centre of the top of every page. EVERY HEADING and EVERY FOOTING cause the same pattern to be placed on each page. If you desire odd-numbered pages to have different headings than even-numbered pages, you can use ODD HEADING and EVEN HEADING.

2.5.2 Including Other Files

There are three ways to have PUB read in other files as part of your document.

```
.SOURCEFILE(filename[ppn]);
.SOURCE!(filename[ppn]);
.SOURCE!L(filename);
```

(where the PPN is optional). The first is intended for most text files; any macros or unusual effects set up in the text file will not affect the file in which the SOURCEFILE macro occurs. SOURCE! and SOURCE!L are intended for including files with macro definitions; they allow effects defined in the files to affect the remainder of the text. SOURCE! expects a full filename and PPN, though the PPN may be omitted if the file is on your own area. SOURCE!L expects to find its file on the standard library area, A700PU00.

2.5.3 Itemized Lists

Occasionally you will have a section of text which enumerates a set of steps, or lists a set of properties. ITEMIZE takes four optional parameters. The macros ITEMIZE and ENDITEMIZE bracket a section of text in which

1. The first line of each paragraph is indented from the current left margin by the contents of the fourth parameter, or the variable INDENT!HEADER if the fourth parameter is omitted. INDENT!HEADER is initially set to 2.

2. The body of each paragraph is indented from the left margin by the contents of the second parameter, or the variable INDENT!LEFT if the parameter is omitted. INDENT!LEFT is initialized to 4.

3. A tab stop is set so that a tab at the start of the first line of a paragraph will align the first line with the rest of the paragraph.

4. The right margin is narrowed by the contents of the third parameter, or the variable INDENT!RIGHT. This causes paragraphs in the list to look thinner than surrounding paragraphs. INDENT!RIGHT is initialized to zero.

Each paragraph in an itemization, including the one just after the ITEMIZE macro, must be preceded by a blank line.

The first parameter to ITEMIZE specifies how the paragraphs in the itemization are to be labelled. If the parameter is present, a counter is set up which is incremented at each paragraph break, and printed at the start of the first line of the paragraph. A "1" in the pattern specifies that the counter is to be printed as an Arabic number; "i", "I", "a", and "A" specify lower and upper case Roman numerals and alphabetics. Extra characters such as parentheses and dots may be included. The above list was produced with the pattern "1.". If the pattern is omitted, then no counting is done.

(a) Note that ITEMIZEs can be nested within each other giving an effect something like this.

> Note that the inner paragraphs are narrowed even further than the outer ones.
>
> This process can continue until the lines get too narrow to hold a single word, at which point PUB will complain.

(b) When you ENDITEMIZE the inner paragraph, everything gets restored properly for the outer one.

This example had an ITEMIZE with no pattern nested inside one with the pattern "(a)". The variables INDENT!RIGHT and INDENT!LEFT were both set to 5 for these examples.

2.5.4 Examples

Two macros, EXAMPLE and ENDEXAMPLE, can be used to bracket sections of text which give programming examples. EXAMPLE goes into NOFILL mode (where lines are neither filled nor justified), indents the text, and sets the A and B fonts to whatever is specified by the variables EXFONTA and EXFONTB. EXAMPLE takes one parameter, the number of characters to indent. If this parameter is omitted, the example is indented by the contents of the variable INDENT!LEFT. It also sets GROUP mode, which means that all of the text of the example will appear on one page. Tabs are set every five spaces; the character "\" (back slash) can be used to space to the next tab stop. ENDEXAMPLE restores things to their previous state.

If you have a large example, it might not fit properly on a single page. There may be several points in the example at which page breaks can occur without disturbing the flow of the example. To let PUB know where you will allow page breaks to happen, you can insert the macro

```
    .hinge
```

at appropriate points.

2.5.5 Tables

The macros TABLE and ENDTABLE can be used to bracket tables. TABLE takes three numbers as arguments, giving the starting positions for each of three columns in the table. It is assumed that the first three columns will be short enough to fit in the available space; the final column may extend across several lines, and subsequent lines will be properly aligned. Each new entry in the table should be separated from the previous one by a blank line. The TABLE macro also goes into GROUP mode, so that the table will appear all on a page if possible.

The tab character, "\" (back slash), should be used to separate columns in the table. The font table in section 2.4.1 was done with TABLE and ENDTABLE.

2.5.6 Figures

These macros provide ways of including XGP image mode files, such as those produced by SPACS, in your document. The intent of these macros is to allow you to define all the properties of the image files in one place (which might be a separate file) and specify the location of the figure at some other time.

The two calls involved are:

```
    .define!figure (tag, size, file)
```

This macro defines the figure, telling PUB how big it is and where find it. TAG is a unique symbol defining the figure. It will become a label that you can use to reference the figure. SIZE is the number of lines required for the figure. SPXIMG will give you this number. Three lines are added for the title. FILE is the name of the image file. Extension ".IMG" is assumed.

The second macro determines where the figure will be placed. The call

```
.put!figure(tag, title)
```

causes the figure to be placed at the point of the call. If there is not enough room for the figure, it will be placed at the top of the next page. TAG is the name used in the DEFINE!FIGURE macro. TITLE is the text to be associated with the figure The text should appear within vertical bars, ||. The figure can be referenced elsewhere in the document using the TAG you specified and the YON macro. The string {YON TAG} imbedded in your text will be replaced by the number of the figure.

For arbitrary floating figures, you can use the macro HEREFIG. HEREFIG takes two arguments, a count of the number of lines in the resulting figure, and something which, when evaluated in command mode, will produce the desired figure. This parameter is usually the name of a macro, but could be any PUB commands. For example,

```
.herefig(6,|group skip 6|)
```

will float a six-line blank figure.

Whether or not the figure macros will be defined is controlled by the variable WANTFIGS. If WANTFIGS is set to TRUE before the macros are loaded, then the figure macros will be defined. Omitting the macros causes the compilation to be faster and take less core.

2.5.7 Indexing and the Table of Contents

If you use the sectioning or indexing macros described above, PUB will prepare a table of contents and an index when it reaches the end of your source file. Entries in the table of contents are made by the sectioning macros (see 2.3). The table of contents will be placed wherever you have said "<u>insert</u> CONTENTS", or at the end of the file if you neglected to say so.

The macros IX and EVAL!IX are used to make entries in the index. IX takes an unevaluated string argument; EVAL!IX evaluates its argument. The macro PRINTINDEX causes the index to be placed at the point of invocation; a PRINTINDEX is automatically done at the end of the input file. Normally an index is always produced if you have any occurrences of the IX macro. You can suppress the index by setting the variable WANTINDEX to FALSE.

2.6. Debugging

The songbook macros have a limited facility for helping you debug large documents with many files, cross-references, and the like. If you add the "U" option to your call to PUB, as COMPILE file.PUB(XU), then USERDEBUG mode will be set. This will cause various debugging information to be included in the output file.

SOS Primer

Joseph M. Newcomer

Introduction

This document is merely intended as an introduction to the SOS editor. For further explanations and a more complete set of commands, consult the SOS manual.

SOS is a Teletype-oriented text editor written by Bill Weiher and Stephen Savitzky of the Stanford Artificial Intelligence Laboratory. It has been maintained and improved locally by numerous people; anyone interested in joining this amorphous group should contact Richard Johnsson, Lee Erman, or Joe Newcomer. Since the current status of the editor changes from time to time, no single document can capture its state adequately. The SOS manual is the basic reference manual for the editor. This manual is an example-oriented primer. The list of changes from the reference manual can be found in the file SYS:SOS.DOC.

In addition to the common editing capabilities of inserting, deleting, and shifting of lines of text, SOS includes string search and substitute commands, an intra-line edit capability, text-justifying features, and a few other assorted bells and whistles.

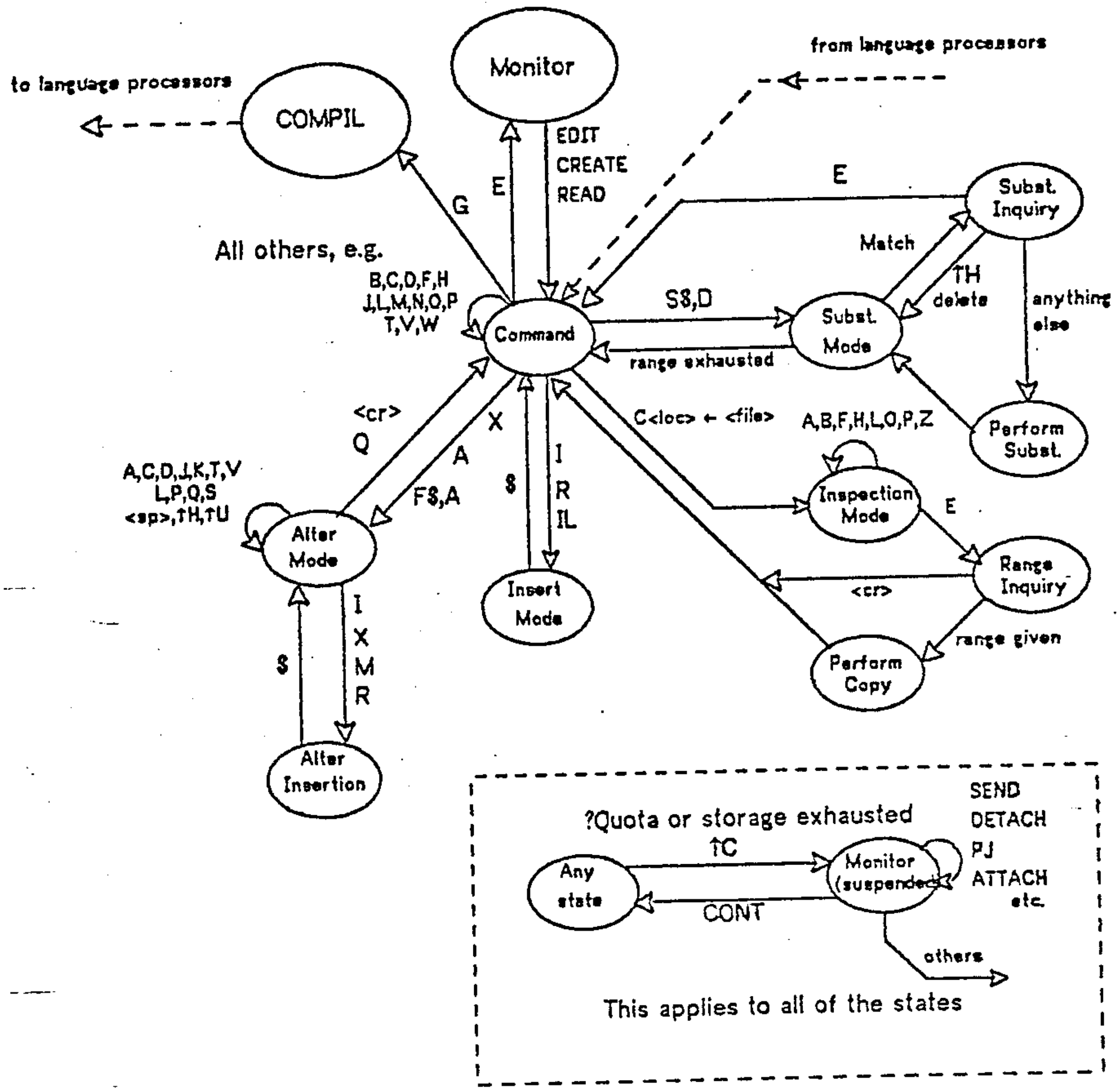
SOS does not edit a file "in place", as some editors do. Changes are made on a temporary copy of the file, and ordinarily are made permanent only upon completion of the edit. However, you may request at any time that all changes up to that point be made permanent. This is an especially recommended practice for beginners, as it insures all changes made in the file since the EDIT command or the last save request against loss due to system failure or user inexperience.

SOS is oriented towards full-duplex devices, such as the Teletype, the ARDS display, the Infoton and Beehive displays, and other such devices. Before attempting to use it from a half-duplex device such as an IBM 2741 terminal or a Datel terminal, you should become thoroughly familiar with using it from the teletype or similar full-duplex device. You must then familiarize yourself with the conventions for using half-duplex devices on the PDP-10 as implemented here at C-MU. In general, it is not worthwhile for the novice to learn how to use SOS from half-duplex devices, since the effort involved in using them does not really make up for the 50% faster typeout (compared to a Teletype).

The HELP facility

When in worry and/or doubt, this command saves a lot of screaming, shouting, and other expressions of confusion. The HELP command (H) will provide you with a quick summary of editor commands. Since the HELP command is self-explanatory (just type H followed by a carriage return) the exploration and discovery of just what this command provides is left as an exercise for the reader.

SOS State diagram



Basic commands

The basic operation in a file-oriented system is the creation of a file. To invoke the editor and request it to create a file, give the CREATE command when the console is in monitor mode i.e., the computer has typed a period.

In all examples, the computer output is underscored.

Example 1 Creating a file:

```
.CREATE BLAT.DOC
00100   THIS IS AN EXAMPLE OF HOW TO CREETE
00200   A FILE USING THE EDITOR.
00300   IN ORDER TO GET OUT OF NUMBERING MODE, TYPE
00400   AN ALTMODE (ESCAPE( CHARACTER, WHICH ECHOES
00500   AS A DOLLAR SIGN.
00600   $
*
```

When the asterisk is typed, you may enter any editor commands you want. The E command (End) terminates the edit, saves the file, and returns to the monitor.

Example 2 Terminating an edit:

```
*E
BLAT.DOC
EXIT
.
```

The file now exists and you may access it in any of the normal modes in which files are accessed. For example, you may type it:

Example 3 Typing a file:

```
.TYPE BLAT.DOC
00100   THIS IS AN EXAMPLE OF HOW TO CREETE
00200   A FILE USING THE EDITOR.
00300   IN ORDER TO GET OUT OF NUMBERING MODE, TYPE
00400   AN ALTMODE (ESCAPE( CHARACTER, WHICH ECHOES
00500   AS A DOLLAR SIGN.
```

If upon examining the typeout, you find there are some errors (as in the typeout above) you may invoke the editor with the EDIT command to make the corrections. The set of commands for simple editing is:

I - Insert

D - Delete

R - Replace

P - Print

O - windOw

L - List

The Replace command is used to replace lines of the file. In its simplest form it is the single letter R followed by the line number to be replaced. The editor then types the line number out and new text may be typed in. This new line replaces the previous contents of the line.

The Delete command is used to delete lines from the file. In its simplest form it is the single letter D followed by the line number of the line to be deleted. The editor deletes the line and returns control with the asterisk. There is normally no other typeout. To delete a group of contiguous lines, a range may be specified; see "Specifying Ranges", below.

The Insert command is used to insert new lines in a file. Its basic format is the letter I followed by the line number of the line to be inserted.

Example 4 Simple editing

```
.EDIT BLAT.DOC
BLAT.DOC
*R100
00100   THIS IS AN EXAMPLE OF HOW TO CREATE
*D400
*I400
00400   AN ALTMODE (ESCAPE) CHARACTER, WHICH ECHOES
*
```

Note that the Replace command has the same effect as a Delete command followed by an Insert command. In order to use Insert to replace a line, the line must first be deleted. The Insert command by itself does not replace the line specified if it already exists, as in some editing systems, but instead creates a new line whose number is equal to the line given plus the line increment (normally 100). The Insert command will always insert a new line in a file, never replace an old one. If the line following the specified line has a line number less than or equal to the computed insertion line number, then the insertion is given a number which is halfway between the line specified and the next line.

Example 5 Interpolated insertion

```
*I200
00250   SINCE THE INCREMENT IS 100, THIS LINE IS HALFWAY
*I250
00275   BETWEEN TWO LINES, AS THIS LINE ALSO IS.
*
```

In order to see what your file now looks like, you can use the Print command to print it on the terminal. The Print command is the letter P followed by the line number of the line to be printed. The letter P by itself will print the current line and 15 following lines. To specify a range of lines, a colon may be used to indicate a beginning and ending line number specification; see "Specifying Ranges", below, for more details on this.

Example 6 Printing part of a file

```
*P100:500
00100   THIS IS AN EXAMPLE OF HOW TO CREATE
00200   A FILE USING THE EDITOR.
00250   SINCE THE INCREMENT IS 100, THIS LINE IS HALFWAY
00275   BETWEEN TWO LINES, AS THIS LINE ALSO IS.
00300   IN ORDER TO GET OUT OF NUMBERING MODE, TYPE
00400   AN ALTMODE (ESCAPE) CHARACTER, WHICH ECHOES
00500   AS A DOLLAR SIGN.
```

The O (window) command is just like the P command except that it prints a few lines before and after the specified line; its syntax is

*O<line>,<count>

The result will be to print <line>-<count>:<line>+<count> as the range; the default if <count> is omitted is 5. This is very useful when you want to look around the line you are currently editing.

In addition to the P command, two keys on the terminal will also cause printing. A linefeed (in this text, ↓), will print the next line, and an altmode (escape, shown as a "$") will print the previous line.

Example 7 Linefeed and Altmode commands

```
*P300
00300   IN ORDER TO GET OUT OF NUMBERING MODE, TYPE
*↓
00400   AN ALTMODE (ESCAPE) CHARACTER, WHICH ECHOES
*$00300   IN ORDER TO GET OUT OF NUMBERING MODE, TYPE
*
```

If there is too much information to conveniently type on the terminal, the L (List) command may be used to output the lines on the printer. Its format is precisely the same as the P command, except that if just "L" is specified the entire file is listed. Note that the file may not come out immediately on the printer, as print files are queued waiting for the printer to become available. Consequently, your file may not be printed for some time after the L command completes. You may continue editing the file, however, since the information is copied into a temporary buffer and held until printed. The file name on the listing printed will be "LINED.LPT". You should not then be looking for a listing with the file name printed on the front.

Example 8 Listing a file

```
*L
*
```

This has printed the entire file on the line printer.

Specifying Ranges

Whenever you wish to specify more than a single line, you may specify a range. This is done by using a colon to separate the two line numbers (where the second must be higher than the first). Thus 100:600 specifies lines 100 to 600. Most commands accept a range of lines to be operated upon, and this is one way of giving that range. However, in some cases it is easier or more appropriate to specify a quantity of lines (5 lines, 17 lines, etc.) regardless of the line number of the last line. This is indicated by using an exclamation point (!) to specify the range: 100!3 is line 100 and the following two lines (so "100!1" is the same as "100").

Example 9 The exclamation point

```
*P100!4
00100   THIS IS AN EXAMPLE OF HOW TO CREATE
00200   A FILE USING THE EDITOR.
00250   SINCE THE INCREMENT IS 100, THIS LINE IS HALFWAY
00275   BETWEEN TWO LINES, AS THIS LINE ALSO IS.
*D250!2
*P100!4
00100   THIS IS AN EXAMPLE OF HOW TO CREATE
00200   A FILE USING THE EDITOR.
00300   IN ORDER TO GET OUT OF NUMBERING MODE, TYPE
00400   AN ALTMODE (ESCAPE) CHARACTER, WHICH ECHOES
*
```

Intermediate commands

The intermediate editing commands are:

C - Copy

T - Transfer

N - Number

W - save World

M - Mark page

G - Go

The Copy command copies lines from one place in the file to another. The first location specified is the "destination" line number. The second location (which may be a range) is the "source" location. The editor will choose an increment which will allow all the specified lines to be copied to the destination without overflowing; this increment is printed out in the message "INC1=nnnnn". If the editor cannot compute an increment such that all lines will fit, then an error message will be typed and appropriate action will be taken by the editor (see the SOS manual, page 13-14).

The Copy command can also copy from another file, so that portions of program files can be extracted to form a new file. Again for details, consult the SOS manual (page 13-14).

The Transfer command is much the same as the Copy command, except that the lines which are copied into the specified destination in the file are then deleted from the source location.

Example 10 Copy & Transfer commands

```
.CREATE COPY.DOC
00100   THIS IS
00200   A SHORT
00300   FILE
00400   $
*C100,300
INC1=00050
*P100:300
00100   THIS IS
00150   FILE
00200   A SHORT
00300   FILE
*T300,100
*P100:400
00150   FILE
```

```
00200   A SHORT
00300   FILE
00400   THIS IS
*
```

The Number command is used to renumber files. This is usually done after a number of insertions have been made and no more room exists between line numbers for further insertions. The simplest form of the Number command is simply the letter N, which renumbers the entire file with an increment of 100. For more information on the Number command, see the SOS manual, page 8-9.

Example 11 The Number command

```
*P100:400
00150   FILE
00200   A SHORT
00300   FILE
00350   THIS IS
*N
*P100:99999
00100   FILE
00200   A SHORT
00300   FILE
00400   THIS IS
```

The W command is particularly useful to the beginner. The W command makes permanent all changes made in the file up to the time it is given. Changes made in a file are temporary until either a W or an E command is given. There are two reasons you should do a W command often: 1) The system could crash, and all editing done would be lost when it came back up, or 2) you might attempt using some new command (say, "substitute", a somewhat tricky one), and confuse your file to the point where you cannot recover the text you started with. In either case, the loss will be back to the last "EDIT" command to the monitor, or the last W command to the editor. By giving permanence to those changes whose accuracy you are certain of, you will avoid losing time in re-creation of those changes, or perhaps the entire file.

SAVE and ISAVE

The one disadvantage of the "W" command is that you must remember to issue it at appropriate times ("You only need to do a W once, but *which* once" is a traditional saying around the department). The ideal point is immediately before the next crash. It is not at all unusual to lose 50 or 60 lines of text if the system crashes at the wrong moment. To prevent this sort of catastrophe, you can as SOS to do a "W" command automatically for you at specified intervals. This is done by use of the SAVE and ISAVE parameters.

When SAVE is set to some number *k*, after every *k* commands which change the file a "W" is automatically performed. When ISAVE is set to some number *p*, after every *p* lines of insertion a "W" is automatially performed.

The SAVE and ISAVE parameters are set with the SOS "Set" command (left arrow) which is discussed in a later section. At this point it is sufficient to know that the commands are "←SAVE=number" and "←ISAVE=number".

Example 11.1 The ISAVE parameter

```
*←ISAVE=3
*I500,5
00505   SOME
00510   TEXT
00515   INSERTED
*
SAVING COPY.DOC
*
00520   HERE
00525   $
*
```

If the system were to crash at the end of the example, lines 505, 510, and 515 would be in the file; only line 520 would be lost.

Pages

Files can be divided into logical subunits termed "pages". A page in the SOS editor is merely a collection of lines. It may be less than one physical printer page, or it may be several physical printer pages. When we need to make a distinction, we will call the SOS pages "logical pages" and the printer pages "physical pages". We will use the term "page" ordinarily to mean a logical page. To indicate the separation into logical pages, a "page mark" is inserted into the file by the Mark page command. The Mark page command places a page mark immediately before the line number specified. Each page is numbered separately, and hence you may have several line 100's in a file. In order to specify what page you are on, use the slash (/) in the line number specification, with the page number following the slash. Line 100 on page 1 is then designated as "100/1". To minimize the amount of typing required, the editor remembers what the current page is, and subsequent commands need only specify the line number on the current page.

Example 12 Multipage file

```
*P100:400
00100   FILE
00200   A SHORT
00300   FILE
00400   THIS IS
*M300
*P100/1:400
00100   FILE
00200   A SHORT
*P100/2:400
00300   FILE
00400   THIS IS
*N.
*P100/1:400/2
00100   FILE
00200   A SHORT

PAGE 2

00100   FILE
00200   THIS IS
*
```

When listed on the line printer with an L command, each page has the page number printed in the upper left. The form of this page number is the logical page number followed by a hyphen followed by the physical page number (recall that logical pages can be longer than physical pages). The physical page number is reset for each logical page, so that the numbers proceed as "1-1, 1-2, ... , 1-n, 2-1, 2-2, ...". When using a listing as a guide to editing, remember that the first number is the page number that SOS uses, e.g. when correcting page 4-15 specify "/4" for the page number.

There are two other special characters which you can use to designate lines in the file. The period (.) is used to designate either the current line or the current page, depending on where it is used. If it is used in the line position, it is the current line; if in the page position it is the current page. If page 2 is the current page, and line 100 is the current line, then "./2" is "100/2", "./1" is "100/1", "200/." is "200/2" and of course "./." is the current line, 100/2. The asterisk is always the last line on the page indicated. If the current line is 100/2 in the file of example 12, then "*" is "200/2" and "*/1" is "200/1". If the line number is omitted but a page number is given, it means the entire page, e.g., "P/2" is the same as "P0/2:*/2". For more details on specifying ranges, see the SOS manual, page 3-4.

Example 13 Period and asterisk designators

```
*P100/1:*
00100   FILE
00200   A SHORT
```

```
*I*/2
00300   NEW LINE
00400   $
*P/2
00100   FILE
00200   THIS IS
00300   NEW LINE
*P*/1:*/2
00200   A SHORT

PAGE 2

00100   FILE
00200   THIS IS
00300   NEW LINE
*P/1:/2
00100   FILE
00200   A SHORT

PAGE 2

00100   FILE
00200   THIS IS
00300   NEW LINE
*P.
00300   NEW LINE
*P100/1
00100   FILE
*I.
00150   INSERTION
*I.
00175   ANOTHER
*P.:*
00175   ANOTHER
00200   A SHORT
*P/1
00100   FILE
00150   INSERTION
00175   ANOTHER
00200   A SHORT
*
```

The Go command is equivalent to the End command in that it terminates the edit; however, it also causes the last COMPILE, EXECUTE, LOAD, or DEBUG monitor command to be re-executed. This is a great convenience when debugging programs.

Example 14 The Go command

```
.CREATE TEST.ALG
            .
            .
            .
*E
TEST.ALG

EXIT

.COM TEST
ALGOL: TEST
200   INCORRECT STATEMENT
REL FILE DELETED
300   UNDECLARED IDEN↑C↑C
.ED
*P200
00200               INTEGRE I, J, K;
*R.
00200               INTEGER I, J, K;
*G
TEST.ALG

ALGOL: TEST

EXIT
.
```

Advanced commands

The advanced editing commands are:

A - Alter

J - Join

IL - Insert Last

S - Substitute

F - Find

B - Beginning

Alter command

The Alter command is one of the most useful features of the SOS editor. It allows editing individual lines much as the normal edit commands are used to edit files. You can alter a single letter in a line, i.e., change it, delete it, or even invert its case. The full capabilities of the Alter command are explained in the SOS manual, page 9 ff; some examples will be given here.

Edit commands in intraline edit mode are not echoed by the teletype. We will indicate this in examples by showing the edit commands in lower case. One exception to this will be the altmode character, which will still be a dollar sign. Remember that in intraline edit mode it will not echo. The following notation will be used: "␣" will be a space, "▩" will be a rubout, "⤶" will be a carriage return, ↑U will be control-U (the control key and U key held simultaneously give ↑U).

The set of intraline edit commands is:

␣ - Accept the character under the pointer

↑H - Backspaces the pointer

▩ - Backspaces the pointer

C - Change the character under the pointer

D - Delete the character pointed to

I - Insert new characters (terminated by altmode)

⤶ - Terminate intraline edit

Q - Quit intraline edit without making changes

↑U - Start over

S - Skip

K - Kill

R - Replace

L - Print remaining line and continue edit

P - Print remaining line and resume edit

V - Invert alphabetic characters

X - eXtend line

M - Munch (Simulate K followed by I)

T - Transpose current and next characters

A - Add characters (a special kind of Insert)

; - Skip to next "column"

If a character is typed which is not a valid intraline edit command; a bell (↑G) is echoed. For explanations of most of these commands, see the SOS manual, pp 15-17. With this as a guide, you may follow the examples below. In these examples, a ↲ is a non-echoed carriage return; a ▦ is a non-echoed rubout, and a ␣ is a non-echoed space.

Example 15 Intraline skip and insert

```
*P/1
00100   FILE
00150   INSERTION
00175   ANOTHER
00200   A SHORT
*A150
00150   seINSi***$↲ERTION
*P.
00150   INS***ERTION
*
```

Example 16 Intraline delete and kill

```
*P150
00150   INS***ERTION
*A.
00150   ssINd\\S↲\\***ERTION
*P.
00150   IN***ERTION
```

```
*A.
00150   s*INkr\\**E↓\\RTION
*P.
00150   INRTION
*
```

You may precede a command with a number which causes it to be repeated, e.g. "2sa" is equivalent to "sa" followed by another "sa".

Example 17 Intraline skip and change

```
*I150
00175   THIS IS A (SMAPLE( LINE
*A.
00175   2s(THIS IS A (SMAPLEc)↓ LINE
*P.
00175   THIS IS A (SMAPLE) LINE
*
```

Example 18 Intraline accept and rubout

```
*P175
00175   THIS IS A (SMAPLE) LINE
*A.
00175   3ssTHIS IS A (2⌣SM■\\M2c\\AM↓PLE) LINE
*P.
00175   THIS IS A (SAMPLE) LINE
*
```

Special features not documented in the SOS manual

The features described in this section are not especially important for beginners, but they are not documented except in the SOS updates. Since users should be aware of them, they are described briefly here.

The character ↑H, control-H, which is equivalent to the "home" key on Infoton terminals, is equivalent to the rubout key. On terminals which physically can backspace, or on video terminals, this allows more readable editing than use of the rubout key. Try it and see.

The V (for inVert) intraline edit command is used when one wishes to change the case of the alphabetic text on the line. The V command without a preceding number (or 0) inverts the case of all alphabetic text starting with the character under the pointer and continuing until the first non-alphabetic character is found. If the character under the pointer is non-alphabetic, then the V command is ignored. If a number is given, then only that many letters starting with the current pointer are inverted, e.g., 1V shifts the case of only one character (handy for capitalizing words). It is awkward to give an example here, since we are using lower case letters in the examples to indicate non-echoed characters. The function should be obvious, however, and easily verified by experiment.

The eXtend command goes to the end of the line and enters Insert mode; this makes it convenient when it is necessary to add text to the end of one or more lines.

The Munch command (so named by Lee Erman and/or Rich Johnsson) can accept a numeric argument preceding it and a single character following it. The command nMx is equivalent to nKxI, i.e., it kills the requisite number of characters then enters insert mode.

The T command transposes the current character and next character, thus correcting in one keystroke one of the most common typographical errors. A number preceding the T command is ignored.

The A command is useful when inserting a fixed number of characters; typically one character. Thus the sequence "Ax" is equivalent to the sequence "Ix$", saving both a keystroke and the necessity of remembering to type the altmode. If a number precedes the A command, that many characters are added before an implicit altmode is performed: e.g. "4Aabcdx" adds the characters "abcd" to the line, then treats the "x" as an intraline edit command (and thus moves to the end of the line and goes into insert mode).

One of the most common errors made in using the Alter command is failure to type the altmode terminating an Insert within the line. This has the effect of terminating the line being edited and beginning a new line. Although a sometimes desired effect, such as in indenting Algol program files, it is more often just an error. Should you type a ↓ after an insertion, and get a new line number instead of the rest of the line, just type the altmode and ↓ again. You now have two lines where you had one before, and the Join command can undo this. To use the Join command, type J followed by the original line number.

Example 19 The Join command

```
*P175
00175  THIS IS A (SAMPLE) LINE
*A.
00175  s)THIS IS A (SAMPLE‿)i OF A↓
00187  $↓ LINE
*P175!2
00175  THIS IS A (SAMPLE) OF A
00187   LINE
*J175
*P175
00175  THIS IS A (SAMPLE) OF A LINE
*
```

The inverse problem exists when you are in normal Insert mode. Typing an altmode kills the current line and terminates the insertion. The usual case is when a large number of alterations have been made you have developed the habit of typing $⤶ to terminate a line. The result is that the line just typed has been lost. To correct this, the Insert Last command exists. Immediately upon returning to command mode type IL, and the last line will be retrieved, put in place, and Insert mode will be resumed (we will not digress into discussing what can be done safely before the line is lost to IL; do it immediately and you will be sure to get the line back).

Example 19.1 The Insert Last command

```
*I300
00300   Text
00400   A terribly (4*R↑2)/(3+(2*X1+X2)) complicated line$
*IL
00400   A terribly (4*R↑2)/(3+(2*X1+X2)) complicated line
00500   $
*
```

Find command

The Find command may be used to locate known strings in a file when their line numbers are not known, or to check a file for occurrences of strings. The basic format of the Find command is the letter F, followed by a string to be searched for, followed by a altmode, followed by a range specification. Again, more details may be found in the SOS manual, pp 23-25. When a string is located, the line containing it is typed out and search is suspended. To resume the search with the same string, only an F followed by an altmode is required.

Example 20 The Find command

```
.EDIT SOME.BLI
*FLOCAL$/1
*
```

(There were no occurrences of "LOCAL" on page 1)

```
*F$/2
00150           LOCAL A, B, C;
*F
00300           LOCAL AARGH BLAT[5];
*F$.+1:/99

PAGE 6

00400           MEASURES LOCALIZED PHENOMENA SUCH AS
*F
*
```

If you give further Find commands without specifying a range, no more strings will be found, since the current line position is the end of the file. To reset the file position, you could either specify the first line of the file as the lower bound of search, e.g., "Fstring$100/1:/999", which is clumsy, or, more simply, you could use the Beginning command to reposition the file. The B command may be given a line number as an argument, in which case it will go to that position in the file, e.g., "B/." goes to the head of the current page, and "B*/.-1" goes to the last line of the previous page.

If you are not interested in stopping at each line where the string is found, you can give a parameter to the Find command which tells how many occurrences to print and bypass before stopping. To find all occurrences in a file, use some large number such as 999 or 99999.

Example 21 The Begin and Find commands

Assume the file is in the state it was left in at the end of example 20.

```
*F
*B
*F$,999

PAGE 2

00150          LOCAL A, B, C;
00300          LOCAL AARGH BLAT[5];

PAGE 6

00400          MEASURES LOCALIZED PHENOMENA SUCH AS
*
```

Substitute command

The Substitute command is similar to the Find command, in the sense that a string is searched for; in addition, a second string is substituted for the one found. The format of the Substitute command is the letter S followed by the string to be searched for, followed by an altmode, followed by a string to replace it, followed by another altmode, followed by a range. For more details, see the SOS manual, pp 25-27.

Example 22 The Substitute command

Assume the file is in the state it was left in at the end of example 21.

```
*B
*SLOCAL$OWN$

PAGE 2

00150          OWN A, B, C;
00300          OWN AARGH BLAT[5];

PAGE 6

00400          MEASURES OWNIZED PHENOMENA SUCH AS
*
```

As you see, the string substitution also replaced the occurrence of "LOCAL" in line 400/6. This is one of the most common errors made with the Substitute command. In this example the Substitute command or the Alter command may be used to correct the problem; in another example it may be neither simple or even possible to undo a bad substitution. For this reason, we recommend giving a W command before doing a Substitute. If the Substitute command then destroys part of the file, abort the edit without making the changes permanent by typing ↑C (control-C), and typing EDIT again. Since you are editing the same file, the file name need not be given.

Example 23 Aborting an edit

Assume the file is in the condition it was in at the end of example 22.

```
*↑C

.EDIT
TEMPORARY EDIT FILE ALREADY EXISTS! DELETE? (Y OR N)
←Y
*P400/6
00400           MEASURES LOCALIZED PHENOMENA SUCH AS
*P150/2
00150           LOCAL A, B, C;
*
```

The message about the temporary edit file may not be typed if the editor was left in a state where the temporary file did not exist.

Miscellany

In addition to the commands discussed here, there are several others of marginal interest. One of the most useful of these is the "=" command, which types out information contained in the editor. Its format is "=" followed by the name of the internal parameter to be displayed. The command is discussed more fully on pp 20-21 of the SOS manual. The most useful parameters to display are the current line (.), the number of pages in the file (BIG), the current FIND and SUBSTITUTE strings (STRING) and the current line increment (INC).

Along with the "=" command there is the complementary "set" command which is a left arrow (←). This is used to change the values of the internal parameters. This is discussed on pp 19-20 of the manual. The most useful parameters to set are the SAVE and ISAVE parameters discussed previously; the next most useful is probably the line increment (INC).

Example 24 The = and ← commands

```
.EDIT HUGE.BLI
HUGE.BLI
*=BIG
62
*P100/41
00100           INCR I FROM 1 TO .N DO
*=.
100/41
*I.,25
00125   BEGIN A←5; X←.Y<3,2>;
00150   $
*=INC
00025
*←INC=5
*I.
00130                   BLAT(); THUD(.Q);
00135   END;
00140   $
*=.
00135/41
*
```

Removing line numbers

In some cases it is necessary to remove the line numbers which SOS places in the file. To do this, you may either have SOS remove the line numbers itself, by specifying the ",N" switch to the End command, or use PIP with the "/N" switch, as shown in the examples below. Note: SOS can edit files without line numbers; during the edit it adds line numbers for reference purposes and then strips them off upon completion. Thus a file without line numbers will continue without line numbers. The user can force line numbers onto a file by editing it and setting DSKNUMS=1; when the edit is complete the line numbers which have been added are retained in the file.

Example 25 Removing line numbers

```
*E,N
BLAT.DOC
EXIT

.
```

or, by using PIP,

```
.R PIP

*/X/N←BLAT.DOC

*↑C
.
```

Example 25.1 Editing a file without line numbers

```
.edit NOLINES.FIL
NOLINE.FIL(,N)
*
  <editing>
*e
NOLINE.FIL(,N)

.
.TY NOLINES.FIL
This file has
no line numbers

.
.edit NOLINES.FIL
NOLINE.FIL(,N)
*DSKNUMS=1
*e
NOLINE.FIL

.
.TY NOLINES.FIL
00100   This file has
00200   no line numbers
.
```

Using terminals with both upper and lower case

Some terminals are available with both upper case and lower case letters, notably the ARDS display, the Infoton video terminals, and the Western Union 300 terminals. The PDP-10 monitor, however, always translates lower case input into upper case unless instructed otherwise. SOS will, whenever possible, determine that the terminal has lower case letters (see the monitor TTY command). If the TTY mode is not correctly set, you can set it by the sequence of commands illustrated below.

Example 26 Using a terminal with lower case

```
.TTY LC
.edit garble.doc
*←m37
*p100
00100   This document describes the GARBLE system of
*
```

Note that when using the WU300 terminals, the "all caps" switch must be turned off, or the terminal will convert lower case letters to upper case letters before transmitting. The TI700 terminals have an "upper case" shift lock in the upper left corner which performs the equivalent function.

When in intraline edit mode, a "skip" or "kill" command will interpret its argument in the exact case it was typed in. Thus in the last example, a skip to "r" from the beginning of the line will stop in "describes", while a skip to "R" will go (from the beginning of the line) directly to the R in "GARBLE".

Using terminals with only upper case

Most terminals available are Teletype model 33 terminals, which have only upper case letters. Occasionally it is necessary to create or edit a file containing both upper case and lower case letters on one of these terminals. SOS allows the case of the input character to be shifted by preceding it with a question mark (?). In normal mode, for example, "A" represents "A", and "?A" represents "a". By changing the mode, "A" will represent "a" and "?A" will represent "A". This is shown in the example below. Note that to get a question mark, two question marks must be typed, i.e., the typein "??" produces the single character "?".

Example 27 Lower case from a teletype

```
.EDIT GARBLE.DOC
*P100
00100   T?H?I?S ?D?O?C?U?M?E?N?T ?D?E?S?C?R?I?B?E?S ?T?H?E GARBLE
 ?S?Y?S?T?E?M ?O?F
*←LOWER
*P100
00100   ?THIS DOCUMENT DESCRIBES THE ?G?A?R?B?L?E SYSTEM OF
*
```

One of the typical problems one encounters when using SOS from a teletype is failure to set mode LOWER, and thus having all the text typed in the opposite case. Because of the frequency of this occurrence, Rich Neely added the V command. The intraline V command (for inVert case) has already been described. Here we will describe the V command as a command.

There are actually three V commands:

VU - Change all alphabetics to upper case

VL - Change all alphabetics to lower case

VV - Invert the case of all alphabetics

Example 28 The V commands

```
*I100
00100   This is some
00200   UPPER CASE and
00300   LOWER CASE text
00400   $
*VU00100
00100   THIS IS SOME
*VL200
00200   upper case text
*VV300
00300   lower case TEXT
*
```

Trouble (and how to get out of it)

This section describes typical problems and how to get out of them with a minimum of pain and data loss, whenever possible.

Problem: ?QUOTA OR STORAGE EXHAUSTED ON str

Cause: There is no more room on the disk structure named for either the temporary edit file or the completed copy of the file (depending upon what you were doing).

Solution: (1) Type "CONT" repeatedly until you can sucessfully gain control and End from the editor. Otherwise your edit will be lost. Do not type any other command (exceptions are noted in the PDP-10 user's manual) as this will probably destroy your core image. Especially do not do a "SYSTAT" of any type.

(2) Type the following sequence to delete files you know to be junk, and resume the edit:

```
.DET
FROM JOB n userid
.LOG userid
login sequence ...
.DEL *.REL,*.LST, etc.  ; delete files you do not want
.ATT n     ; reattach to edit job
FROM JOB m userid
.CONT
```

When you sucessfully complete your edit (immediately; do not wait) be sure to re-attach to job m and kill it (K/F).

(3) If you are not actually editing, but are only examining pieces of a file, specify the /R switch (meaning "readonly") to the EDIT command, e.g., EDIT file/R. No temporary file will be created, and hence no disk space required. Any commands which would change the file are considered illegal.

(4) Yell and scream. This is often the most effective method because someone in the terminal room will begin to feel guilty and go delete some useless files, thus giving you enough space to continue on.

Problem: Accidental typing of ↑C

Solution: Type "CONT". Almost any other command will destroy your core image (for exceptions, see the PDP-10 user's manual).

Problem: Sudden return to monitor from intraline edit

Cause: You have accidently typed ↑C in intraline edit mode. It is not echoed, so there is no direct evidence that this is what happened.

Solution: Handle the same as any accidental typing of ↑C, by typing "CONT". You should then type a carriage return (<cr>) to return to the editor command mode. This minimizes confusion about where you are in the line.

Problem: Carriage return in ALTER mode keeps giving new line numbers instead of terminating the ALTER.

Solution: Type an altmode followed by a carriage return. You have been in intraline insert mode. Use the Join command (J) to glue together the broken line (see example 19).

Problem: *OUT OF ORDER*

Cause: A line number is out of sequence, i.e., lower than its predecessor. This may be caused in several ways:

(1) A Copy or Transfer command for which an increment of 1 was insufficient, e.g., copying 7 lines to the space between lines 200 and 205.

(2) A Justify command (which renumbers the lines) which created more lines than originally existed (you should set INC=3 or some other small value before justifying lines to avoid this problem).

(3) Accidental or deliberate removal of a page mark ("D/n" removes the page mark for page n).

Solution: Renumber either the entire file (N command) or the offending page (N,/m for page m). If a page mark was accidently deleted, you can now go back and put it in, and renumber the newly-created page.

Problem: Typed an altmode on a line in normal Insert mode.

Solution: Use the IL command to retrieve the lost line. See example 19.1.

Problem: Lower case letters print with question marks (instead of printing in lower case).

Solution: Set mode M37 (←M37).

Problem: *LINE TOO LONG*

Cause: A line longer than 177 characters was found while reading the file. (Note: question-mark characters which print as two characters still require only one on the line).

Solution: R SOSBIG, a special version of SOS with extra-long buffers. Otherwise this version is the same as SOS. SOSBIG cannot be called with the EDIT command.

Problem: System crash.

Solution: Cry. Then resolve to set SAVE and ISAVE the next time! Note: It is occasionally possible to recover some or all of the editing lost. Look for a file nnnSOS.TMP (where nnn was you last job number, e.g., job 4 creates a file 004SOS.TMP). Then do the following:

```
.REN HOPE.TMP=nnnSOS.TMP
.R FILCOM
*TTY:←HOPE.TMP,input.fil
```

i.e., you compare the contents of HOPE.TMP with the contents of the file you were editing. If HOPE.TMP seems to be more complete, then continue to put changes in it, instead of the original file. NOTE: do not delete the original file until you are absolutely certain that HOPE.TMP is correct! The reason that the RENAME command is used is that some people use SOS to look at their temp file instead of using FILCOM. If they happen to have the same job number it is obvious that this method fails.

Patterns in Find and Substitute

This section may be ignored by most beginners. It is included in this primer because a good explanation (with examples) does not appear anywhere else. There are more details in the SOS manual, pp. 22ff.

Certain special characters may be included in the search string for Find and Substitute commands. Rather than being interpreted as characters, they are interpreted as requests to match a class of characters which they designate. These characters are (as described in the SOS manual, and repeated here for convenience):

Char	Escape	Function
|	?:	Will match any "separator", i.e., anything not a number, a letter, or the symbol period, percent, and dollar sign (., %, and $).
∀	?/	Matches any character.
¬	?%	Not the next character, i.e., inverts the match. Thus A¬BC matches all strings of the form AxC, for "x" any character except B. ¬| will match any letter (not-separator). The special sequence ¬∀ matches either the beginning or the ending of a line.
∞	?)	Means "any number of" the next character (from 0 to "infinity").
≡	?7	Quotes the next character (this is what you use to search for one of the above characters, e.g., to find a vertical bar you must say "F≡|", not "F|" (which finds the first separator).

We will now consider an example. In all examples, we will not print the line numbers, but you may safely assume they are present. You may have done an EDIT /N, for example. In all cases, the "formal" pattern will be given, followed by the escape-character equivalent in parentheses. Only the graphic display processors (GDPs) have the full SOS character set available at the keyboard. On all other devices you will have to use the escape conventions. In the first example, we have a file of names and addresses. We wish to locate all people who are listed with only initials for their first names, for example, so we can change this to their full first name. Our input file is:

```
J. Smith
5075 W. Ave.
Gorp, Pa., 16111

H. Q. Bovik
4150 ScH Hall
Pittsburgh, Pa., 15213

Alan J. P. M. Shaft
4150 ScH Hall
Pittsburgh, Pa., 15213
```

We already know that the sequence ¬| (?%|) will match any letter, so we try the command "F¬|.$,999" ("F?%|.$,999") with the following results:

```
J. Smith
5075 W. Ave.
Gorp, Pa., 16111
H. Q. Bovik
Pittsburgh, Pa., 15213
Alan J. P. M. Shaft
Pittsburgh, Pa., 15213
```

Obviously our search string was too general. It matched any sequence of a letter followed by a period. Upon examining the file, we find that all first initials start at the beginning of a line, so we use the special beginning-of-line sequence ¬∀ (?%?/) to help us in the search. For the command "F¬∀¬|." ("F?%?/?%|.") (literally, "the beginning of the line, followed by a single not-separator, followed by a period"), we get:

```
J. Smith
H. Q. Bovik
```

The patterns may become arbitrarily complex. If they become too complex for SOS to handle, it will complain; this does not usually occur.

Now that we know we can find a certain piece of data, let us change it. This example is contrived in the sense that it probably does not make any sense for a real file, but it is pedagogically simple. We wish to replace the first initial of a name by the string "***INITIAL***". For this we can use a Substitute command. The first argument to the Substitute command is the "old string"; actually, it is used to Find the old string. The second argument is the "replacement string", in this case "***INITIAL***".

```
*B
*S¬∀¬|.$***INITIAL***$                    (S?%?/?%|.$***INITIAL***$)
***INITIAL*** Smith
***INITIAL*** Q. Bovik
*
```

This may be useful in flagging all the initials in some obvious way so that they can be seen on a printout. However, it does destroy the actual initial. In order to preserve it, we can use the "partially specified string" indicator in the "replacement string" portion. The specification ∂n∂ (?*n?*) indicates that the nth partially specified string in the "old string" portion of the command be used for the indicated portion of the replacement string. Thus the command "S|.$∂1∂!!!DOT!!!$" (S|.$?*1?*!!!DOT!!!) will replace all occurrences of a separator followed by a period by the same separator (∂1∂ [?*1?*] is the string which matched the |) followed by the string "!!!DOT!!!". Our only anomaly, which this example was specifically contrived to point out, is that the partially specified string ¬∀ (?%?/) must be counted. Thus the actual initial can be specified in the replacement string as ∂2∂ (?*2?*) . Thus we get

```
*B
*S¬∀¬|.$***INITIAL***"∂2∂"$
                                   (S?%?/?%|.$***INITIAL***"?*2?*"$)
***INITIAL***"J" Smith
***INITIAL***"H" Q. Bovik
```

LIST OF EXAMPLES
(See index for page numbers)

INDEX

C. D. Councill with script by T. Teitelbaum

TECO is the "text editor and corrector" provided by Digital Equipment Corporation (DEC) for creating and editing files recorded in ASCII characters on any standard device. It is an interesting anomoly since it can serve as a simple editor, but can also be used as a programming language. It can perform simple editing functions as well as sophisticated search, match and substitute operations, and operate upon arbitrary length character strings under control of commands which are themselves character strings (and can exploit this recursiveness). When one knows how to use it, TECO can be extremely versitile. It is not, however, everyman's editor.

At present, TECO on the PDP-10 has no maintainer; TECO, since it is the major editor, on C.mmp is maintained by Eric Woudenberg. On the PDP-10, there is very little on-line documentation for TECO. In fact, the only document, at present, consists of a list of the changes made to TECO by past CMU maintainers of the program. DEC publishes both a TECO reference manual and an introductory manual which are available at the PITT Bookstore if you truly wish to learn this editor/language.

Below is a script written by a past TECO enthusiast which shows some of the uses and methods of TECO.

SCRIPT

TECO

TECO is a text editor. The text being edited is stored as a single character string in the TECO buffer. This buffer is always just as long as the string it contains. The boundaries of the buffer cells are numberd starting to the left of the first character with zero. The index of the boundary to the right of the last character is known as "Z". Thus, the buffer containing the string "ABCD" may be pictured as:

Z

```
-----------------
| A | B | C | D |
-----------------
0   1   2   3   4 = Z
```

M,N A subfield of the buffer is desgnated by the integer pair, "M,N", where M<N. Thus, in the example above, the subfield "1,3" currently contains the string "BC". We may refer to

H the whole buffer by "h" which is really just an abbreviation for "0,Z".

<CR>   Text in TECO has no line numbers, unlike SOS or LINED. The RETURN key of the terminal is treated like any other symbol, with the exception that it is input to the buffer as the two characters "carriage-return" and "line-feed". Thus, the line:

```
                ABCD<CR>
```

will appear in the buffer as:

```
                ---------------------------
                | A | B | C | D |<CR>|<LF>|
                ---------------------------
                0   1   2   3   4    5    6 = Z
```

.   Associated with the buffer is a cursor which can be moved to point to places of insertion, deletion and so forth. The current boundary position of the cursor is known as ".".

*   TECO signals that it is waiting for commands by typing a "*". Arbitrarily many commands may be strung together in a command

$$ string which is terminated by two altmodes (ESC on some terminals). Note that the altmode echos as a "$". On receiving the $$ TECO will interpret the command string from left to right, then will return to the user for more by typing a "*".

Let us now use TECO to create a file called SCRIPT.TEC. Remarks added after or during the session will appear intermittently and will be indented.

We will enter from PDP-10 monitor mode with the command "MAKE". This is used when a new file is being constructed.

```
.MAKE SCRIPT.TEC

*2=$$                           What is the value of "2"?
2
*.=$$                           Where is the cursor?
0
*.=Z=$$                         Where is the cursor and where is
0                                the end of the buffer?
0
*.=$Z=$$                        An altmode ($) between commands is
0                                optional to improve clarity.
0
*HT$$                           Type the whole buffer.  It's empty.
*IABCD                          Insert the line "ABCD" and
$HT$$                            type the whole buffer.  The text of
ABCD                             the insertion stops at the first altmode ($).
*.=Z=$$                         Where is the cursor and the end of
                                 the buffer?
6                               Cursor is after the last insertion.
6                               Buffer is 6 long (remember two for <CR><LF>).
*IEFGH                          Insert some more lines.  Insertion is
IJKL                             always made at the point of the cursor.
MNOP
$$
```

```
*HT$$                          Type the whole buffer.
ABCD
EFGH
IJKL
MNOP
*J.=$$                         Move the cursor to the beginning of
0                               the buffer.
*CC.=$$                        Advance the cursor two.
2
*-2C.=$$                       Move the cursor back two.
0
*6D$$                          Delete six characters to the right of
*.-$$                           the cursor and left adjust string
0                               in the buffer.
*ZJ-6DHT$$                     Jump the cursor to the end of the
EFGH                            buffer, characters to the left and
IJKL                            type the whole buffer.
*0,6KHT$$                      Kill the subfield between 0 and 6.
IJKL                            Note that 0,6D won't work.
*HK$$                          Kill the whole buffer.
*IONE                          We insert some lines so that we can
TWO                             exhibit the line oriented commands.
THEE
$$
*-2T$$                         Type the previous two lines.
TWO
THEE
*-LT$$                         Move the cursor back a line and
THEE                            type one line.
*CCT$$                         Move the cursor forward two characters
EE                              and type the rest of the line.
*IR$$                          Insert the correction.
*0LT$$                         Return the cursor to the beginning
THREE                           of the line and type the line.
*LIFOR                         Advance the cursor a line and
FIVE                            continue inserting.
$$
*JSFO$T$$                      Jump to 0 and search until "FO".  Note
R                               that the cursor is placed after the
                                pattern is found.
1*IU$0LT$$                     Insert the correction and type the line.
FOUR
*ISIX                          Continue text insertions.
SEVEN
EIGHT
$$
*HT$$                          Type the whole buffer.
ONE
TWO
THREE
SIX                            We forgot to move the cursor before this
SEVEN                           insertion and so it was misplaced.
EIGHT
FOUR
FIVE
```

```
*JSSIX$0L.=$$                 Use search to place the cursor at
17                             line "SIX".  Type the cursor position.
*3LT$$                        Place the cursor three lines down.
FOUR
*17,.XA$$                     Save from 17 to . in register A.
*17,.K$$                      Delete same subfield in buffer.
*ZJGA$$                       Jump the cursor to the end and get
*HT$$                          (insert) register A.  Type the whole
ONE                            buffer.  Ahh, that's better.
TWO
THREE
FOUR
FIVE
SIX
SEVEN
EIGHT
*-2K$$                        Delete the previous two lines.
*ZJ-2T$$                      Assure that the cursor is at the end
FIVE                           and type the previous two lines.
SIX
*EX$$                         Exit.  This will write out the buffer
EXIT                           to the opened file "SCRIPT.TEC".
.                              and return us to PDP-10 monitor mode.


.TECO SCRIPT.TEC              Editing existing files is done with a TECO
                               command which fetches the first few
                               characters.
*1000<A>$$                     A backup file (e.g. SCRIPT.BAK) is
                               also made.
*HT$$                         The remainder of the buffer is filled
ONE                            using the APPEND command.  Values
TWO                            greater than 1000 may be needed for
                               large files.
THREE                         Make sure your buffer is full by typing
FOUR                           it or the last few lines of it.
FIVE
SIX
*J5<S                         Here 'specific' iteration is used to
$-2DI $>$$                     change the first 5 occurrences of
*HT$$                          carriage return/line feeds (<CR><LF>)
                               to blanks.
ONE TWO THREE FOUR FIVE SIX    The commands in the brackets are
                               repeated as many times as is SPEC.
*J<S $;-DI                    'Arbitrary' iteration (indicated by the
        $>$$                   absence of a number and the presence of
*HT$$                          a ;) iterates until there is no
ONE                            match, then the brackets are exited.
   TWO
      THREE
           FOUR
               FIVE
                   SIX
```

```
*J5<S
     $-DI
$>$$
*HT$$                         Type the whole buffer.
ONE
TWO
THREE
FOUR
FIVE
SIX
*J<SO$;0LT$L>$$               A frequent use of iteration is to
ONE                            "print all occurrences".
TWO
FOUR
*HT$$                         Type the whole buffer again.
ONE
TWO
THREE
FOUR
FIVE                          Interpretation of this command string
SIX                            is left as an exercise to the reader.
*0UA$J<S
$;-2C$.-QAUB$QC-QB"LQBUC'$.+2UA$L>$0UA$J$QC+1UC$<S
$;-2C.-QAUB$QA+QC+2UA$0L$QC-QB<I $>L>HT$$
   ONE
   TWO
 THREE
  FOUR
  FIVE
   SIX
*EX$$

EXIT                          Finally, a return to monitor mode.
```

C-MU PDP-10 TECO QUICK REFERENCE GUIDE

---Help Sheet Conventions---

↑n	means control character n.
↥n	means uparrow then character n.
#	means not equal to.
↑RUBOUT	rubout or delete character.
(sp)	space.
$	ESC or ALT depending on terminal.

---Initialization and File Selection---

dev:filnam.ext[ppn]	File specifications
ERfilespec$	Select file for input
nEM	Position magnetic tape
EZfilespec$	Zero directory and select file for output.
EWfilespec$	Select file for output.
EBfilespec$	Select file for input and output, with backup file protection.
MAKE filespec$	Same as EWfilespec$.
TECO filespec$	Same as EBfilespec$ Y$.

---Input---

Y	Input one page if buffer empty.
A	Input one page and append to current buffer contents.

---Buffer Positions---

B	Before first character; 0.
.	Current pointer position
Z	Number of characters in the buffer.
m,n	m+1st thru nth characters in buffer.
H	Entire buffer; B,Z.

---Argument Operators---

m+n	Add.
m n	Add.
m-n	Subtract.
m*n	Multiply.
m/n	Integer divide.
m&n	Logical AND.
m#n	Logical OR.
()	Perform enclosed operations first.
↑O	Accept number in octal radix.

---Output and Exit---

nPW	Output the current page and append a form feed character to it n times.
nP	Output the current page, clear the buffer and read in the next page. Do this n times.
m,nP	Output the m+1st thru nth characters, don't add form feed or change buffer.
EF	Close the output file.
↑Z or ↑Z	Close output file, exit to monitor.
↑C	Exit to monitor.
EX	Output remainder of the file, close the output file, exit to monitor.
EG	Same as EX except execute the last compile class command issued.

---Pointer Positioning---

nJ	Jump pointer to position between nth and n+1th characters.
nC	Advance pointer n positions.
nR	Move pointer back n positions, =-nC.
nL	Move pointer n lines.

---Deletion---

nD	Delete n characters following pointer.
-nD	Delete n characters preceding pointer.
nK	Delete from pointer thru n lines.
m,nK	Delete m+1st thru nth characters.
Wtxt$	Delete from . to start of txt in buffer.

---Insertion---

Itxt$	Insert "txt" into buffer.
nI$	Insert an ASCII n (decimal).
@I/txt/	Insert "txt" delimited by character following I into buffer.
n\	Insert the ASCII representation of the decimal integer n.
↑V	Translate to lower case.
↑W	Translate to upper case.
↑↑	When used in text arguments this means translate special characters @,[,\,],↑,← to "lower case" range.
↑R	Accept next character as text.
↑T	Used in text arguments to cause all control characters except ↑R, ↑T, and $ to be taken as text. Nulled by second ↑T.

---Type Out---

nT	Type next n lines of the buffer.
m,nT	Type the m+1st thru nth characters.
n=	Type the decimal integer n.
n==	Type the octal integer n.
1ET	Change typeout mode so no substitutions are made for non-printing characters.
0ET	Restore typeout mode to normal.
0EU	Flag lower case characters on typeout.
1EU	Flag upper case characters on typeout.
-1EU	No case flagging on typeout.
-1ES	Set automatic typeout after searches.
nES(n>0)	Set automatic typeout after searches and include a character to indicate the position of the pointer.
0ES	Set to no automatic typeout after searches.
↑Atxt↑A	Type the text enclosed by ↑A.
↑L or ↥L	Type a form feed.
↑O	Inhibit typeout.

---Aids---

0EO	Restore the EO value to standard.
nEO(n#0)	Set the EO value to n.
1EH	Type only code part of error messages.
2EH	Type error code plus one line.
3EH	Type all three parts of error.
0EH	Equivalent to 2EH.

C -MU PDP-10 TECO QUICK REFERENCE GUIDE

---Search---

nStxt$	Search for the nth occurrence of "txt", do not go beyond the page end.
nFStxt$txt$	Do an nStxt$ then replace found string with second string, don't go beyond the page end.
nNtxt$	Same as nStxt$ except pages will be output and input till "txt" is found.
nFNtxt$txt$	Same as nFStxt$txt$ except that an Ntxt$ type search is done.
n←txt$	Same as nNtxt$ except pages are not output, but thrown away. (be careful!)
:nStxt$	Like nStxt$ except return -1 if successful or 0 if failure instead of an error message. Can also be used with FS,N,FN, and ←.
@nS/txt/	Like nStxt$ except that "txt" to be searched for is delimited by character that follows S. May be used with FS,N,FN, and ←.
0↥X or 0↑X	Reset search mode to accept either case.
n↥X or n↑X (n # 0)	Set search mode to "exact" mode.
↑V	Translate to lower case.
↑W	Translate to upper case.
↑↥	Same as when used with Insert
↑R	Accept next character as text.
↑T	Same as when used with Insert.
↑\	Used inside search arguments to indicate accept either case for following characters. Nulled by second ↑\.
↑X	Used in a search string, match any character at this position in the found string.
↑S	Match seperator character at this position.
↑Na	Accept any character except the arbitrary character a at this position.
↑Q	Take the next character in the search string literally, even if it's a control character.
↑EA	Accept any alphabetic character as a match.
↑EV	Accept any lower case alphabetic character as a match.
↑EW	Accept any upper case alphabetic character as a match.
↑ED	Accept any digit as a match.
↑EL	Accept any end-of-line character as a match.
↑ES	Accept a string of spaces or tabs as match.
↑E<nnn>	Accept the ASCII character whose octal value is nnn as a match.
↑E[a,b,c...]	Accept any one of the characters in brackets as a match.

---Q-Register---

nUi	Store the integer n in Q-register i.
Qi	Equal to the value stored in Q-register i.
%i	Increment Q-reg i by 1 and equal this value.
nXi	Store in Q-reg i, text from the pointer to the nth line.
m,nXi	Store m+1st thru nth characters in Q-reg i.
Gi	Place the text in Q-reg i into the buffer.
Mi	Execute text in Q-reg i as a command string.
[i	Push contents of Q-reg i onto the Q-stack.
]i	Pop contents of Q-stack into Q-reg i.
*i	(As first command in a string.) Save the preceding command string in Q-reg i.

---Iteration and Flow Control---

n<>	Do the enclosed command string n times.
n;	If n>=0 jump out of the current iteration.
;	Jump out of the current iteration if the last search failed.
!tag!	Define a position in the command string with the name "tag".
Otag$	Jump to the position defined by !tag!.
n"Ecmnds'	If n=0, execute commands between E and '; otherwise skip to the '.
n"Ncmnds'	If n#0, execute enclosed commands.
n"Lcmnds'	If n<0, execute enclosed commands.
n"Gcmnds'	If n>0, execute enclosed commands.
n-1"Lcmnds'	If n<=0, execute enclosed commands.
n+1"Gcmnds'	If n>=0, execute enclosed commands.
n"Ccmnds'	If n is the ASCII value (decimal) of a symbol character, execute 'cmnds'.
n"Dcmnds'	If n is a digit, execute 'cmnds'.
n"Acmnds'	If n is alphabetic, execute 'cmnds'.
n"Vcmnds'	If n is lower case alpha, execute 'cmnds'.
n"Wcmnds'	If n is upper case alpha, execute 'cmnds'.
n"Tcmnds'	If n is true, execute 'cmnds'.
n"Fcmnds'	If n is false, execute 'cmnds'.
n"Scmnds'	If n is successful, execute 'cmnds'.
n"Ucmnds'	If n is unsuccessful, execute 'cmnds'.

---Special Numeric Values---

1A	ASCII (decimal) of character after pointer.
↥E or ↑E	form feed flag, =0 if no form feed was read on last input, otherwise -1.
↥N or ↑N	End of file flag, =-1 if end of input file seen on last input, otherwise 0.
↥F or ↑F	Decimal value of console data switches.
↥H	Time of day in 60th's of a second.
n↥H	n=1 DEC date. n=2 then -1 if in PROFILE.
ET	Typeout mode switch =0 norm, -1 otherwise.
↥X or ↑X	Search mode flag =-1 exact, 0=either case.
EU	Value of EU flag. =1 Flag upper case chars, 0 flag lower case chars, -1 don't flag.
EO	Value of EO flag =1 for v21a, 2 for v22&v23.
EH	Value of the EH flag. =1 code only, 2 code plus one line, 3 all of error message.
↥↥x or ↑↥x	Equal to the ASCII value (decimal) of the character x following ↑↥.
\	Equal to the decimal value of the digit string following the buffer pointer.
↥T or ↑T	Stop command execution and then take on the ASCII value (decimal) of the character typed in by the user.

---Aids---

/	After error to type out detailed message.
*i	Used at the beginning of a command string, this causes the entire command string to be moved into Q-register i.
?	After an error, types the bad command.
?	Enter trace mode. A second ? command takes TECO out of trace mode.
↑RUBOUT	Erase the last char typed in command string.
↑G↑G	Erase the entire command string.
↑U	Erase everything back to the last linefeed.
↑G(sp)	Retype current line of command string.

XCRIBL---A Hardcopy Scan Line Graphics System for Document Generation*

R. Reddy, W. Broadley, L. Erman, R. Johnsson, J. Newcomer, G. Robertson and J. Wright

In certain areas of computer science research, conventional line printers and graphics terminals have proven to be inadequate output devices. Typical problems such as a display of digitized (speech or visual) data require either displaying a very large number of (flicker-free) vectors or simulating gray scale output. The need for a hardcopy computer output device capable of producing arbitrary type fonts, graphics, and gray scale images has been obvious. The XCRIBL system, developed at Carnegie-Mellon University (CMU), using a Xerox Graphic Printer (XGP) driven by a minicomputer represents an inexpensive solution to the problem. Careful design of data structures and interface permits the minicomputer to generate each scan line for the XGP as needed without having to resort to brute force solutions. Although the XGP was designed over ten years ago, it had not found wide acceptance as a computer output device because of the excessive processing time and memory requirements of scan-line generation.

The XGP is a facsimile copying machine originally designed for transmission of documents over high bandwidth telephone lines. It has adjustable resolution; the one described here is operated at 192 points per inch which is equivalent to an image of approximately 3.5 million bits for an 8½×11 page. Because of its high resolution each page can contain information equivalent to two pages of conventional computer listing. The XGP printer is a synchronous device, requiring a complete raster line every 5 milliseconds. In order to make the project economically reasonable, a decision was made to use a low-cost minicomputer, a Digital Equipment Corporation PDP-11, with a 28k (16 bit) memory. The limited computing power of the machine influenced many design decisions, such as the inclusion of "modes" of operation of the interface.

The usual Xerox process consists of reflecting light from a printed page onto a selenium drum. The change in electrical charge on the drum caused by the light is used to transfer the "toner" to paper, where a high temperature "fuser" makes the image permanent. Instead of reflected light, the XGP uses the image generated on a cathode-ray tube, one scan line at a time. The image on the CRT is produced by facsimile transmission or, in this case, under computer control. The image is transferred to unsensitized 8½×11 inch continuous roll paper at a speed of 1 inch/second; the paper may be cut to size automatically under computer control.

The PDP-11/XGP system operates as a peripheral device to the main computer, a PDP-10. The character set descriptions for various type fonts may be stored on a

*This research was supported in part by Xerox Corporation and in part by the Advanced Research Projects Agency of the Department of Defense under contract no. F44620-70-C-0107 and monitored by the Air Force Office of Scientific Research. We would like to thank Bill Gunning, Dave Damouth, and Louis Mailloux of Xerox Corporation for their help and assistance.

small head-per-track disk connected to the PDP-11, or kept on the PDP-10. Text and graphic information are transmitted as needed from the PDP-10 across a high-speed data link (160,000 bits/sec). In addition to textual and graphic information, the data from the PDP-10 may also contain special purpose control information such as changes of type fonts, variations in margins, and special formatting requests such as line justification.

An interesting feature of the system is that every aspect of the output device now becomes a variable when compared with conventional line printers. The character sets, size, all margins, interline spacing, and page size are all variable, and can be changed dynamically during the output of a document.

Representation of Information

Characters are represented internally as a rectangular bit matrix. Each row of the matrix requires an integral multiple of 8 bits (the byte size of the PDP-11), although not all the bits of the last byte may be used. Characters may be any width from 0 to 255 bits wide and (theoretically) up to 2^n-1 bits high.

Vectors are represented in a conventional scan line format. This format is necessarily different from the ordinary representation of vectors, since for most graphics terminals the entire screen is randomly accessible. In video terminals and hard-copy scan line devices the data must be presented in the order that the scan lines are generated. A software solution to the problem of vector intersection with scan lines was chosen in order to retain the capability for flexible formatting of the output. Vectors are processed in real time, and the available computing power limits the number of vectors which can cross any scan line.

Gray scale representation is achieved by dividing the page into 1/25 inch squares (an area of .0016 square inches) in which an appropriate number of bits is set to black to represent darkness. This is achieved at present by using a rectangular spiral representation of increasing darkness. Generation of gray scale images thus turns out to be a special case of textual output in which a special gray scale type font is used.

The generation of a scan line which contains both textual and graphic information is not a problem for the PDP-11 if the text and graphics is non-overlapping. If the latter is not the case, then one has to resort to an off-line solution of generating the bit image on the PDP-10 or restricting the character set to only fixed-width characters. This is a restriction in the present system but may not be permanent.

IMPLEMENTATION

In this section we provide a description of the overall implementaion of the system. More detailed descriptions of the various aspects of the system may be found in [1].

The Interface

The purpose of the interface between the PDP-11 and the XGP is to accept a coded scan line from the PDP-11 memory and decode it into a video signal, every 5 milliseconds. A scan line is a bit vector of about 1550 points, in which each point is either on (black) or off (white). There is no gray scale available at this level. The interface has facilities for handling three different modes of data and means for switching between modes, as well as providing control and interrupt functions. The modes available are "character mode", "vector mode", and "image mode".

In the character mode, the first byte sent to the interface represents the number of valid bits (and consquently, the number of following bytes) which contain the data. When the width count is given as zero, then the next byte represents a mode change (to either vector mode or image mode) or a stop code, indicating completion of the data.

In the vector mode, each pair of bytes represents a run-coding of (part of) the data. The first byte of the pair represents the number of white points and the second byte represents the number of black points. When two successive bytes are zero, the interface reverts to character mode.

In image mode, every bit is treated as video information until an error condition occurs, typically "overscan", at which point an interrupt is caused for restart of the next scan line. Because of the high data rate required, this is the only mode which cannot operate in real time from the PDP-10; for this mode, the scan line images are first sent to the PDP-11, where they are accumulated on the disk before being transferred to the XGP.

The support system

There are two components to the support system; one resides in the PDP-11; the other operates as a user program in the time-shared PDP-10. The purpose of the PDP-11 support system is to generate the scan line data needed by the XGP. The support system also services interrupts from the PDP-10/PDP-11 link, examines the incoming data for control information, and selects type fonts from the disk as needed. All of this is done subject to the real-time constraints of the XGP.

The part of the support system which resides in the PDP-10 provides the users with the facilities of sending text, vectors, and character sets across the link. It also provides for conversion of vectors from conventional format to scan line format.

The Character Set Design System

BILOS is a system for the creation and modification of character sets and has many facilities that are common to other interactive editing systems. Rather than manipulating lines of text, BILOS manipulates the rectangular bit matrices which define characters. Any bit of a character matrix may be set or reset by moving a cursor to the appropriate point on a grid and issuing a command.

In addition to these manipulations, the system has facilities for copying, substituting, translating, rotating, stretching, shrinking and reflecting characters. The system currently runs on a storage screen display terminal connected to the PDP-10.

Document Generation Languages

The XGP provides a powerful and flexible tool for the production of printed documents. Since there is a very low cost associated with producing a copy of a document, the user is free to experiment with type fonts, typographic style, physical arrangement of the text and illustrations, etc., until the desired document is produced. The flexibility of type fonts allows mathematical or technical notation to be used freely, without the necessity of typing or drawing the symbols on the final document. Furthermore, the output is "camera-ready"---a distinct advantage in light of rising publication costs.

Two languages for text preparation exist on the PDP-10 at CMU -- XOFF and PUB. Both have been modified to interface with the XGP and are documented in manuals available from the Computer Science Department.

INTRODUCTION TO LOOK

LOOK is a PDP-10 program which transmits information from the 10 to the PDP-11 controlling the XGP. Complete documentation of look is available on file LOOK.DOC[A730GR02]. Below is the sequence of commands used to print this document on the XGP. User input is underlined, comments in lower case.

```
.R LOOK

*!OUTA NGR25.KST          file name for the a partition character set

*!OUTB NGRU25.KST         file name for the b partition

*←NL=55                   set the number of lines per page to 55

*XCRIBL.XGO               name of the file to be printed

*↑C
```

Using the XGP

Philip Karlton,9-Aug-76

The Xerox Graphic Printer (XGP) is the local name for the marraige of a Long-Distance Xerography (LDX) device with a PDP-11/45 with support software. The PDP-11 is a dedicated processor; and, even though it is capable of being a PDP-11, it is rarely used to do anything but drive the XGP. The XGP can move paper and make black marks on that paper upon the request of the PDP-11. From now on, we will refer to the whole thing as XGP.

The basic paradigm for getting something printed on the XGP is to run a program called LOOK on the PDP-10/B. LOOK does the handshaking with the XGP to ship files and commands accross to it. If the files you wish to print are on the PDP-10/A, then run the program BOOK and give it the list of file names you need. BOOK will ship all the files to the /B over the ARPA net and will run LOOK for you. your terminal will then be logically on the /B and you are free to give LOOK the commands you wish. (It is possible to get the /A to talk directly to the XGP by switching the patch panel near the PDP-11 console. This is highly discouraged. If the /B is down go ahead and consider doing it. The link from the /A to the XGP is about one hundred times slower than the one from the /B.)

LOOK is now ready to accept commands. The simplest command is just a file name (default extension .XGO). Once you finally get to talk to the XGP, LOOK will inform you of the progress of the shipment of your file. Your file at this point is stored on the fixed head disk of the XGP. The printing now starts and your pages will be curled indiscriminantly into a cardboard box in front of the XGP.

There are a large number of state variables and commands available in LOOK. See the XCRIBL documentation for all the gory details. PUB and XOFF (which is no longer supported but is much faster) are capable of producing .XGO files that take advantage of many of the esoteric features of the XGP.

Some of the simpler commands:

```
←AKSET=<character set> or <number>
←BKSET=<character set> or <number>
!OUTAKSET <file name>
!SHIP <file name>/<number>
```

Explanations follow. A large number of character fonts are available on the system. Most of them exist on [A730KS00]. Copies of some of these are kept on the XGP disk. The XGP System supports (unfortunately only) two character sets (A and B) in core simultaneously. The default AKSET is NGR25 and the default BKSET is empty. <file name> above refers to a file containg a character set definition. (Character sets can be edited using BILOS.)

OUTAKSET replaces the A character set with one defined in the file. SHIP puts the character set on the disk under the access tag <number>. Nothing will be transfered by the SHIP command if a character set <number> exists on the disk. Storing fonts on the disk is handy if you plan to be switching back and forth between character sets without the need to transfer the character set each time. Superfluous transfers will not be done; pick some <number> in the range 200 to 15000 that some one else will not pick.

@SYS:PROGRM is not really one single command. It causes LOOK to execute the comands in the file SYS:PROGRM.XMD. The result is that the state is appropriate for listing source code of programs.

In order for the XGP to print your file it must be shipped first. There are many dragons in the way of getting your file safely out onto hard copy. First, you must get the link to the XGP. If someone else is using it, you will have to wait until she is done. LOOK will inform you that it is waiting on getting the link. Once you have the link, you must wait for the currently printing file to finish being printed. LOOK says nothing to you while it is doing this. Unfortunately, this is the same thing it says if the XGP is not running. If you cannot figure out what the state is, ask! Let me repeat that: ASK!! After you have watched the magic process several times, you will become the expert and will be able to pass on the proper incantations. It is possible to reload the XGP System by either typing L on the console TTY if the PDP-11 is still running or by restarting the PDP-11 at location 173000. Errors such as "no response from high-speed link" do occur and they should not frighten you or make you feel that you have broken something. Ask someone around you what to do. If she can't help you, then both of you go and ask yet another person. Eventually you will find an answer.

1. C.mmp Algol 68 System

1.1. Overview

The initial Algol 68 system on C.mmp accepts a PAGE, UNIVERSAL of PAGEs, or SUPER FILE as input, compiles this source text and invokes the run-time system which executes the program. Eventually this compile-and-go system will give way to a compile-link-and-go system which supports separate compilation and program libraries.

1.2. Using the System

The Policy Module 0/Hydra/C.mmp system has much in common with the "Turing Tarpit" where everything is possible and nothing is easy. The terminal user interface, called the Command Interpreter (CI), allows each user to tailor the interface to conform to the particular user's deep-seated beliefs and light-hearted whims concerning terminal interfacing. Some important features which support this flexibility are the macro and procedure (called COMMAND object) definition facilities of the command language, a general directed-graph directory structure, automatic invocation of a user profile (a designated COMMAND) at login time, and, in general, the ability to do anything in any of hundreds of ways.

In order to spare Algol users some of the pain normally encountered when setting out to use Hydra, a set of COMMANDs and macros, as well as a simple profile have been developed. The following subsection describes how to get started on C.mmp using these aids. The next subsection is aimed at the experienced Hydra user and describes, in Hydra-ese, the primitives available to anyone operating in I-roll-my-own mode.

1.2.1 How to Do It (for beginning Hydra users)

Obtaining an account. If you do not have an account on Hydra, a request for one should be made to the operations staff. This normally means asking Carolyn Councill or the PDP-10 operator to get an account for you.

Logging in. After giving the front-end the command to connect your terminal to C.mmp, one of four things may happen. One, you may be told the host is down. Try again later. Two, the prompt character "@" may be typed. Good. You are communicating with the Job Monitor (JMON) and may proceed. Three, nothing happens. To get some attention, press the BREAK key (Control-K). You should receive an innocuous message followed by the prompt. Four, something else happens. You are on your own.

Assuming you have made it to the prompt, type: CL. Some chatter will appear on your terminal and eventually it will get back to a new prompt, ">". Now you are communicating with the CI; type: LOG(). The login dialogue will be self-explanatory. If your user directory contains a COMMAND in an entry with the name Profile, this COMMAND is invoked as the final step of the login process.

Creating a profile. PMO is nearly unusable without an appropriate user profile. If you do not have a profile, create one by typing the following supplication: &sysdirectory.Public.Algol68.CMDs.GetProfile(). The resulting profile contains macros which help implement the commands described in the remainder of this section.

Preparing source input using SOS. A version of the SOS editor exists under PMO. The PDP-10 SOS Manual serves as the user manual for the PMO editor. However, only a subset of PDP-10 SOS is available. In particular, the Copy, Transfer, Find, Substitute and Justify commands are not implemented.

A new file may be created and edited using SOS by typing: Create(). You will be prompted for a file name. Your reply should be a name of one to ten letters and/or digits. (Other characters may be used, but some are rather dangerous.) SOS starts in insert mode.

An old file may be edited using SOS by typing: Edit(). Your reply to the prompt for a file should be the name given when you created the file or a null reply (i.e. carriage return) in which case the file most recently edited or compiled will be used.

SOS may be used to examine a file in read-only mode by typing: Read().

Additional information about PMO SOS may be found in the file SOSC.DOC[A110LC00] on the PDP-10A. This information is of little use to a beginner. Problems with and complaints about the PMO editor should be sent via MAIL to A110LC00 on the PDP-10.

Preparing source input using TECO. A version of the TECO editor exists under PMO. It is roughly a subset of its namesake on the PDP-10. A new file may be created and edited using TECO by typing: Make(). The prompt for a file name should be answered with a name consisting of one to ten letters and/or digits. An old file may be edited by typing: Teco(). You will be prompted for a file name. A null response causes the last file edited or compiled to be used.

Specifying files and file names. If you have created any programs using SOS or TECO, your directory now has an entry called Algol. This is itself a directory, and there is an entry in it for every program you create. When you use one of the standard commands, such as "Edit" or "Teco", and it prompts you for a file name, it looks up that file name in your Algol directory. If you want to deal with a file that isn't in your Algol directory, you can do that, too; the only requirement is that you specify completely, using the conventions of the CI, how to access the file. For instance, if your user directory has an entry called Test, and Test is a directory with an entry called Prog which is your program, then when "Edit" or "Teco" or whatever prompts you for a file name, you should type: Test.Prog. If your program isn't even on a directory, but is in a "Capability variable" called &prog, then you should type: &prog.

You can bypass the prompt by passing the proper indication of your program directly to the command, as a parameter. For instance, you might type:

```
Edit(Algol.prog)
   ! Equivalent to:
   ! Edit()
   ! Source file: prog
Edit(Test.prog)
   ! Equivalent to:
   ! Edit()
   ! Source file: test.prog
Edit(&prog)
   ! Equivalent to:
   ! Edit()
   ! Source file: &prog
```

Notice that these commands "remember" what the last file you edited, ran, etc. was. That is, if you respond to the prompt by not typing anything but carriage return, it is as if you typed the name of the last file you edited or ran. The commands do this by maintaining a "Capability variable" called &currentfile. At any given time, this variable is set to the file currently in use, or the last file that was in use. Ordinarily you don't need to know this, but for the curious, that's how it's done.

Every directory entry (indeed every object in the Hydra system) has a type. Files created by SOS are of type SUPER FILE; files created by Make() are of type UNIVERSAL. You'll notice that the type of each file is printed out when you get a directory listing (see below) of all your files. SUPER FILE cannot be edited with TECO, and UNIVERSALs cannot be edited with SOS, in case you were wondering.

Compiling and executing a program. First, type: Alg68(). The following prompt will be typed:

```
            Source Input:
```

The response should be the name of the file to be compiled and executed. The dialog continues with the prompt

```
            Listing Device:
```

This is the first of several "option prompts" which are part of the standard dialog. For each option prompt, there is a set of possible replies, and if you forget any of these, you can have the system type out the whole set, by replying with "?". Any reply may be abbreviated to only its first few characters, enough to distinguish it from all other replies in the set. A reply must be followed by a carriage return.

Currently there is only one possible reply to the 'Listing Device' option prompt. This is "TTY", indicating that a compilation listing is to be typed at the user's terminal. A null reply (i.e. a bare carriage return) indicates that the listing is to be suppressed. In the future, more replies (e.g. "LPT") will be available.

Next, there is another option prompt:

```
            Compiler Option:
```

This prompt is repeated after each reply, until the user types a null reply (carriage return). A list of the available replies may be found in Section 1.2.3. After the null reply to this prompt, the compilation begins.

When the compilation is complete, a message stating the program's code and data sizes (in number of words) is typed on the terminal. If no compilation errors occurred, the run-time system is called. First, however, there is another option prompt:

Runtime Option:

Like the previous option prompt, this one is repeated after each reply, until the user types a null reply. A list of the available replies may be found in Section 1.2.4. One of the replies is special: 'STANDOUT'. This brings on the 'Standout Option' option prompt, by which means the user specifies what is to be the nature of the standoutchannel, the channel on which the standard output file is opened. The available replies are:

LIST	It is a file of the LPT subfile type, which is listed as soon as it is closed.
TYPE	It is the same as Consoutchannel, the channel for output to the user's terminal.
SAVE	It is a file of the SOS subfile type. More about this later; this is the default.
DELETE	Not implemented yet.

The run-time system then greets the user with some reassuring message and commences program execution. After the program finishes, another reassuring message appears at the terminal indicating that execution is complete. If the user has earlier specified the 'SAVE' Standout option (or has not specified an option--this is the default), the accumulated output from the program using standoutchannel is now available as a file of the SOS subfile type, and the system gives the 'Final Standout Option' option prompt. You may specify that the file be typed on your terminal, listed on the line printer, saved in your userdirectory under the name 'Standout', or thrown away (it is biodegradable).

If you run a program more than once without editing it, you can shorten the compilation-execution sequence considerably, by avoiding more than one compilation. To do this, first type: Com68(). You should do this whenever you have done some editing and are ready to try out the program again. This runs the Compiler, which produces an Object Program from your program. If you have a subdirectory called Object on your userdirectory, the Object Program will be put there; otherwise it will be put on your userdirectory. To actually run this program, type: Run68().

Terminating Algol 68. If your program gets into an infinite loop, or some other mishap befalls it, press the BREAK key (Control-K). Ordinarily, this will cause the program to be interrupted and forcibly stopped. You can also interrupt the compiler this way, if you want to.

Listing a file. Type: List(). You will be prompted for a file name, and for a print name, which is the name that gets printed on the banner page of the listing.

Typing a directory. To see your user directory, type: Di(). To see some other directory, such as Algol, type: Di(Algol).

Deleting a file. If your program is called "prog", type: del(algol.prog).

Logging out. First, type KJOB to the CI. After it calms down, you should be talking to JMON again. (You may have to type another carriage return to get the "@" prompt.) Right now, it is necessary to type KJOB to JMON as well. (In the future, that won't be necessary.) That wasn't so bad, now was it?

1.2.2 How to Do It (for battered Hydra users)

For more detailed information on how to use the facilities described in the preceding section, the advanced Hydra user can explore the contents of the Public.Algol68 directory; in particular, Public.Algol68.CMDs which contains the various COMMAND objects.

1.2.3 Compilation Switches

Some of the compilation options listed below are not of interest to users; these are marked with an asterisk. Of the others, currently all are available as pragmat-items as well. For instance, if your program contains the pragmat

```
:: lower ::
```

the remainder of the program (at least up to the next pragmat) will be assumed to use the "lower" stropping convention.

DEBUG*	Enable compiler debugging mode of operation.
GHOST*	Mumbo jumbo.
LISTING	Produce compiler source listing output.
LOWER	Use lower stropping convention.
NAKED*	Hocus pocus filiocus.
NODEBUG*	Disable compiler debugging mode of operation.
NOGHOST*	Disable mumbo jumbo.
NOLISTING	Suppress compiler source listing output.
NONAKED*	Disable hocus pocus filiocus.
NOWARNINGS	Do not output warning messages.
POINT	Use point stropping convention.
RES	Use reserved word stropping convention.
UPPER	Use upper stropping convention.
WARNINGS	Output warning messages.

1.2.4 Execution Switches

DEBUG	Enable run-time system debugging mode of operation. This is of some limited usefulness to users. The user is prompted for a set of 'Flags', that is, a number; this number is treated as if it had been passed as the second argument to a call on Systrace (see documentation elsewhere) in the program.
PROCESSES	The user's program is to be run on more than one process. The user is prompted for a number of processes, which may be from 1 to 16. Note that this is not a number of physical PDP-11 processors, but of HYDRA processes; and that at any time the user's program may make use of fewer processes than the specified number, or it may be written as if it could make use of more; in either case it should still run and produce correct results.
SPEEDUP	Consult the implementors before using this switch.
STANDOUT	See the explanation in Section 1.2.2.

1.2.5 Error Reporting

A compilation error is indicated by the printing of a vague message, the line containing the error, and a line containing a position indicator beneath the first character of the most recently seen input lexeme at the time of the error. A position indicator is simply a digit printed in the appropriate position. If more than one error occurs in a given line, the error messages, first to last, are associated with the position indicators, left to right, respecting the following rule. If many errors occur at the same position, the digit printed indicates the number of errors which occurred there.

Syntax errors will normally cause some portion of the input to be ignored. For the most part, any ignored characters are printed on the listing with equal signs beneath them.

No semantic processing of the program is done after the first compilation error occurs. Naturally, the run-time system is not called if an error is found.

There are also abnormal situations which are not quite as severe as errors. These cause a warning message to be printed in the same format as an error message except the fact that it is a warning is noted. Warnings do not cause semantic processing to stop.

Run-time errors are normally indicated by the printing of an accurate statement of what the problem is. This is accompanied by an indication of the source program line which was being executed when the error occurred.

REFERENCES

1 Algol 68 User Manual, HYDA68.XGO[L150AL72]/A

2 Revised Report of the Algorithmic Language Algol 68, SIGPLAN Notices, May, 1977.

3 Pagen, Frank, A Practical Guide to Algol 68, John Wiley & Sons, London, 1976 (Available in CMU Bookstore)

4 Tanenbaum, Andrew, "A Tutorial on Algol 68", Computer Surveys, June, 1976

few valuable writings on C.mmp/Hydra that exist:

1. Proceedings of the 5th Symposium on Operating Systems Principles, Austin, Texas, November 1975. Section VIII has three papers that form the best introduction to C.mmp and the Hydra Kernel.

2. The Hydra Songbook. This vigilante users' tool was first initiated by Brian Reid and has developed with the contributions of Hydra's many users. A new edition will be out during Autumn 1976. The previous edition is now about six months old and many of the specifics in it are wrong. It is especially valuable to the Bliss-oriented user; others will find the documentation of the Command Language the songbook's most valuable part.

3. Hydra User's Manual. This manual supplies the detailed reference for the Hydra Kernel. Joe Newcomer will have a new edition out during September 1976. A very valuable tool.

4. C.mmp Algol Reference. Describes the Algol68 implementation available on C.mmp. This document should be used with the informal introduction to the Algol68 sublanguage.

1. TECO

A version of TECO, a subset of the PDP-10 program, exists to run under Hydra. It can edit a virtually unlimited amount of core (over 100 pages). It edits a universal object with pages in it.

There are commands objects in &SYSDIRECTORY.UTILITIES which can interactively invoke TECO.

```
TE()
for pages or
EDITCMD()
 for commands objects.
```

To create a new universal and put text into it, use the following sequence of commands:

```
CAPA &name
$MAKEUNIVERSAL(&name)
&SYSDIRECTORY.PROCEDURES.TECO(&ttyport,&name)
```

You may now edit happily away.

1.1. Hydra-teco commands

The command language of this teco is very similar to that of Teco on the PDP-10 however only a subset of the features exist. In this light only a quick description will be given of the commands which are identical.

Further information may be found by consulting:

1) PDP-10 USERS HANDBOOK
2) CMU Introduction to the PDP-10
3) CMU PDP-10 Teco quick reference guide

These commands are identical on the C.mmp and the PDP-10:

B	0, buffer origin.
Z	character count of the buffer, buffer end.
H	B,Z
nC	skip n characters.

nD	delete n characters.
E	exit.
nFSab	find "a" and substitute "b" (n finds, 1 substitute).
Ia$	insert text "a" into the buffer.
nI$	insert ascii n into the buffer.
nJ	jump to the n'th character in the buffer.
nK	kill n lines.
m,nK	kill characters between buffer position m and n.
nL	skip n lines.
nR	skip n characters backwards.
nSa$	search for the n'th occurence of "a".
nT	type n lines.
m,nT	type characters between buffer position m and n.
;	leave iteration if preceding S or FS fails.
n<>	do commands enclosed in <> n times (forever if no n).
n==	type the value of n in octal.
n=	type the value of n in decimal.
n+m	the value of n + m.
n-m	the value of n - m.
↑R	quote the following character, in this way <ESC> and ↑C can inserted and searched for.
#	like the pdp-10 TECO ↑O meaning take the following number as octal.

In addition the S and FS commands will take a negative argument causing them to search backwards. A backwards S (-S) will leave the buffer pointer at the beginning of the string it finds. If a search fails the buffer pointer is left where it was, as in SOS.

1.2. Q-registers

Teco has q-registers which behave almost identically to those in PDP-10 teco. Internally there are actually two sizes of q-registers but these are completely transparent to the user. This was done in the interest of speed.

There are 36 q-reg's: the letters A-Z and digits 0-9.

The commands available for q-registers are:

nUm	store the integer n into q-reg m
Qm	equal to the integer stored in q-reg m
nXm or i,jXm	store the next n lines into q-reg m or store characters between buffer positions i and j into q-reg m
Gm	put the text in q-reg m into the text buffer at the current pointer position.
Mn	execute the text in q-reg M as a teco command string. For now the user must be careful not to execute an 'I' command with a string longer than 8192 characters. The insert command has not yet been modified to take longer strings.
*m	as the first command in a command string. Store the last command string in q-reg m. this is a loss recovery technique.

1.3. extended features

There are some commands in C.mmp Teco which do not exist on the PDP-10, these are as follows:

1P,	type out a map of your text pages at the screen top after every $$
P	type out the page map and turn off page map typeout.
↑W	must be used to "fool" the command interpreter into allowing this pdp-10 construct:

```
FSa$$
FSa$↑W$ must be done instead.
```

Note that this is only necessary in an iteration.

Three SOS-like commands exist:

<↑A>	as the first character in a command string is equivalent to the teco command string '-LT' or go to the previous line and print it.
<LF>	as the first character is equivalent to 'LT' or go to the next line and print it.
<CR>	as the first character is equivalent to '0LT' or go to the beginning of this line and print it.

1.4. implementation details

The command input buffer is 4000 characters long. Teco will warn you when you are less than 150 characters from the end. If you do not heed this warning by typing **$$** (executing the command) execution will begin automatically after reading the 3999th character. Then any characters that were ignored as a result of getting input line at a time will be printed.

On exit, teco writes the byte count of the buffer size into the first two words of the data-part of the universal object it edited, least significant word first. Note that teco does not use these numbers itself but places them there for the benefit of other programs.

If ↑K↑K (or ctlprocess(&pros,-1)) is typed it will have various effects depending upon what teco is currently doing. The possible actions are:

a) During typeout: return to command mode.
b) At the beginning of any command execution: return to command mode.
c) Anywhere else: no action is taken.

The inner search loop in teco can search for a character it will never match at approx. .2 seconds per page. This includes time spent shuffling pages.

Introduction to the CMU Graphics Terminals

Sam Harbison & Steven Rubin, 11-August-77

The Computer Science Department has a number of graphics terminals designed and built by our engineering lab. Their full name is "Graphics Display Processor", or GDP. The terminals can be recognized by their large CRT screens and green keyboards.

The graphics terminals can be used as regular terminals to the PDP-10s, as well as for more sophisticated graphics applications. General information about the structure of the GDP is given here to help you orient yourself. Programming details are given in other documents listed at the end of this introduction.

Each graphics terminal is attached to a PDP-11 which interfaces to the PDP-10. This PDP-11 runs a small operating system known as the "graphics monitor", which can also run special user programs that use the graphics cpability of the terminal. Be aware that not all graphics terminals are equivalent. Some have more powerful PDP-11s with more memory, drawing tablets, or other special equipment.

Under normal circumstances, all characters typed at the terminal are received by the graphics monitor which sends them unchanged to the PDP-10, where they are interpreted the same as characters from any terminal. Likewise, text from the PDP-10 is passed directly to the GDP display by the graphics monitor. The point here is that the terminal will be completely inoperative when the graphics monitor is not running. When the monitor stops, it can be restarted from the PDP-11 console switches.

The graphics monitor, in addition to passing characters between the terminal and the PDP-10, has several useful utilities available to the terminal user, including an intra-line editor which interfaces to SOS. The use of these facilities is discussed later.

The GDP's PDP-11 can also execute user programs. This feature allows real-time programs which drive the graphics processor to execute locally so as to not tie up the PDP-10. User programs may be created, compiled, and linked on the PDP-10, and then loaded into the PDP-11 and started by PDP-10 monitor commands or under PDP-10 program control. User programs executing on the GDP's PDP-11 may communicate with the terminal's keyboard/display, with the graphics monitor, and with the PDP-10.

By convention, certain characters (called "meta-characters") typed on the terminal keyboard are assumed to be destined for PDP-11 software (the monitor or a user program) and are not passed to the PDP-10. The use of these characters is discussed later.

Many people have written interesting programs for the graphics terminals. Some of the more useful ones include:

SPACS - a program which allows the user to interactively compose complex pictures via the drawing tablet. The finished pictures can be stored on the PDP-10 and printed on the XGP.

BILOS - a program which allows the user to build character sets for the XGP.

SPACEWAR, STAR-TREK, PETAL, LIFE - some of the demonstration/game programs written for the GDP. See GDP.DEM[A630GS00] for a list of the many demonstration programs available.

Using the GDP2

The following information is excerpted from the Graphics Monitor Manual, written by Steven Rubin and Donn Bihary. This document is available in the document room if you want more information on the GDP. See also the reference list at the end of this section.

Use as a Terminal

The GDP can be used to communicate with the PDP-10 much like any other display terminal. The user should keep in mind, however, several differences:

1) The keyboard is attached to the PDP-11, and only if the PDP-11 monitor is working can communication to the PDP-10 be established.

2) The GDP has a Graphics keyboard, which has more keys than a normal keyboard, and is thus used slightly differently.

3) The presence of programs on the PDP-11 makes it useful to designate certain keyboard characters (the Meta characters) to be read by the PDP-11 software and not passed on to the PDP-10.

4) The processing power of the PDP-11 makes possible certain features (e.g. the intra-line editor) not possible with "non-intelligent" terminals.

These differences are examined on the following pages.

System Start-up

If the PDP-11 monitor is inoperative, it may be reloaded via the PDP-11 console switches. Starting the PDP-11 at address 173000 will load the monitor and a character set from the PDP-10. Note that this type of restart, called a bootstrap restart, will abort any running program on the PDP-10 and start another to reload the PDP-11. Minor system crashes can sometimes be fixed by a soft restart at address 1004, or a hard restart at address 1000. If DDT (the debugging package) has been loaded, it can be restarted at address 1002 (if DDT is not present, this address will do a hard restart). These do not affect the PDP-10. All of the above restarts can be done from the keyboard provided that the PDP-11 is running (see the subsection on Meta Characters).

The Graphics Keyboard

The Graphics keyboard has two types of keys: four **special keys** marked Shift, Top, Control, and Meta, and the fifty-eight other keys (called **encoded keys**). The special keys operate like the Shift key on an ordinary teletype; when depressed they cause no immediate action, but they change the interpretation of any encoded keys struck while they are depressed.

The special keys usually have the following interpretations: Shift is used only to get upper-case alphabetic characters. (The Shift Lock switch can be used to generate all upper case letters.) The Top key is used to get the upper symbol which appears on the encoded keys. (NOTE: The Shift key does this on most other terminals, but on this keyboard Shift applies only to alphabetic characters.) The Meta key indicates that the character is to be interpreted by the PDP-11 software and not passed to the PDP-10. The Control key is used like Control on teletypes, i.e., it indicates a special control function to be performed, not necessarily a printing character.

Throughout this manual, the term Meta Character will be used to denote the character generated by striking an encoded key while the Meta key is depressed. Individual characters in this group are denoted Meta-A, Meta-B, etc. Likewise we have Meta-Control characters when both Meta and Control are depressed. (In this case, Meta is dominant, i.e. the character is interpreted by the PDP-11.) And, of course, we can talk about Top characters, Control characters and Shift characters. All of these are keys that can be held down while striking an encoded key. Top-Shift characters are just Top characters - the shift is ignored. Top and Shift are usually ignored if Meta is depressed, although the PDP-11 programs can (and sometimes do) find out if they were depressed.

A few of the encoded keys are by their very nature Control characters, and it is not necessary to depress Control to use them as such. The keys, with their Control character equivalents, are given below.

Key	Equivalent	Function
BREAK	Control-S	Stop output
CLEAR	Control-Q	Restart output
FORM	Control-L	Form feed
VT	Control-K	Vertical tab
RETURN	Control-M	Carriage return
LINE	Control-J	Line feed
CALL	Control-C	Interrupt program
TAB	Control-I	Tabulation

Meta Characters

As mentioned, Meta characters and Meta-Control characters are not sent to the PDP-10. They are used to specify parameters or request action by PDP-11 software. Because it may be necessary to use Meta characters to communicate with several different parts of the monitor as well as various user programs, a convention has been adopted on the use of Meta and Meta-Control characters to avoid confusion.

Meta-Control characters are used to put the keyboard in various modes. Each mode is generally associated with one feature of the monitor or user program. (For instance, a mode to control the scroller, a mode to control the intra-line editor, etc.) Each mode has associated with it a group of **Meta characters** which causes certain actions. (For example, Meta-S can be used to set the size of the scroller text when the keyboard is in scroller mode.)

Changing the keyboard mode with Meta-Control characters associates specific actions to various Meta characters. The same Meta character may have different effects depending on what Meta-Control mode is currently in effect. User programs may establish their own Meta-Control characters (with associated Meta characters). Note that there are special Meta characters that never change in meaning. They are:

Meta-BREAK	Call DDT. If there is no DDT loaded, this has no effect.
Meta-\	Soft restart. This is useful for stopping runaway programs since it returns control to the monitor yet does not harm the user program or the display.
Meta-CALL	Hard restart. The monitor destroys everything except itself. Useful if your program is hopelessly tangled.
Meta-*	Bootstrap restart. The PDP-10 is called in to reload the monitor (this takes about 30 seconds). Necessary if your program wiped out both itself and the monitor.
Meta-Z	Clears the keyboard number (see next subsection)
Meta-DASH	Indicates negative keyboard number
Meta-0, ..., Meta-9	Entering keyboard numbers to the Meta-routines

At start-up time, all Meta characters are set to have no action (except for the special Meta characters) and the following system Meta-Control characters are available:

Meta-Control-S	Scroller control
Meta-Control-K	Keyboard control
Meta-Control-E	Intra-line editor
Meta-Control-G	Graphics control (not described here)
Meta-Control-I	Input/output control (not described here)

The Meta characters associated with these modes are presented later.

Keyboard Numbers

We mentioned before that Meta characters could be used to set parameters in the system. It is thus necessary to have some way to specify numerical values for the parameters. TECO and SOS users know that often numbers can be associated with one-letter commands by typing that number just before the command. A similiar general feature is available in the GDP system.

Typing a series of Meta digits prior to a Meta character will cause the number to be passed to the Meta routine that processes the character. Some Meta routines interpret the number as octal, others as decimal.

There are two other Meta characters used with the keyboard numbers: Meta-Z clears the keyboard numbers (useful when you make a mistake while entering the numbers), and Meta-DASH causes the following number to be negated.

For example, suppose you want to set the number of characters on a line to 50. You read in this manual that Meta-C in Scroller mode (set by Meta-Control-S) sets the number of characters on a line and that it uses a decimal value. Thus you type Meta-Control-S, Meta-5, Meta-0, and Meta-C. The Meta-Control-S puts you in scroller mode, Meta-5, Meta-0 sets the keyboard number to 50 (or 40 if read in octal but the octal value will be ignored) and Meta-C causes the number of characters to be changed.

Scroller Meta Characters

Normally, when using a GDP as a terminal, a user types commands to the PDP-10 and the PDP-10 executes them. This dialog is distinguished as a series of text lines on the screen. The text is scrolled by the graphics monitor in the same way as a normal terminal: the top line disappears forever when a new line is needed at the bottom and there is no room. The monitor does, however, support multiple scrollers which can be selectively displayed for multiple conversations and extended screen capacity.

This set of commands (entered with a Meta-Control-S) control the scroller. Commands that require a parameter use the decimal keyboard number unless otherwise indicated. If multiple scrollers are desired, the number of alternate scrollers should precede the Meta-Control-S. Thus, to create 19 additional scrollers (for a total of 20 scrollers) type Meta-Control-1-9-S. You may create up to 99 additional scrollers. It is not advisable to change the number of additional scrollers without doing a hard restart (Meta-CALL) but you may type Meta-Control-S with no parameter at any time and it will not change the number of scrollers.

The first three Meta commands below affect the selection and control of the scrollers. Note that at all times there is a "current" scroller for which the remainder of the Meta commands apply and on which all PDP-10 conversation appears.

Meta-P	**Page set**, Set the current scroller to the keyboard number. Stop displaying the previous scroller and start displaying the current scroller.

The keyboard number must be from 0 to the maximum number of scrollers defined by the Meta-Control-S.

Meta-G	**Get page**, Set the current scroller to the keyboard number and display this scroller. The previous scroller is not turned off so many of them may be displayed with this command.
Meta-O	**Off**, Stop displaying the scroller determined by the keyboard number. This does not affect the current scroller.
Meta-A	**Scale**, Set scale to octal keyboard number. This may range from 0 (quarter scale) to 17 octal (3.5 times normal). The normal scale is 10 octal. See Appendix C for other scale values.
Meta-I	**Intensity**, Set intensity to octal keyboard number. The intensity may range from 0 (which you can't see) to 17 octal (which is the normal intensity).
Meta-X	**X position**, Set X-position of upper-left margin of the scrolled characters (initial value -475).
Meta-Y	**Y position**, Set Y-position of upper-left margin of the scrolled characters (initial value 475).
Meta-L	**Lines**, Set number of lines that can be displayed in the scroller (initially 56).
Meta-C	**Characters**, Set number of characters per line (initially 98).
Meta-U	**Units**, Set number of tab units per line (initially 98). Note that this parameter helps control the number of characters per scroller line. The characters parameter is simply the number of characters that may appear on a line. The units parameter determines the number of possible locations that a tab may start from. For the standard fixed width characters set, the number of units equals the number of characters. When using variable width character sets, characters may end anywhere on a line and considerably more tab characters are needed to line up the tab stops.
Meta-J	**Jump**, Sets the number of extra lines to jump when the scroller fills and must be rolled up (initially 0).
Meta-S	**Scale**, Set the scale parameter as in Meta-A, but also set the **Lines**, **Characters**, and **Units** parameters to values appropriate for the new scale. This command is the easy way to change the size of scrolled text.
Meta-N	**Normal**, Reset all of the above parameters to their default values. For the standard fixed width character set, **Scale** is set to 10 (octal) **Intensity** is set to 17 (full on) **X position** is set to -475. **Y position** is set to 475.

Lines is set to 56. **Characters** is set to 98. **Units** is set to 98. **Jump** is set to 0.

Meta-FORM	Clear all scrolled lines except the current line.
Meta-CLEAR	Like Meta-FORM, except that the current line is also cleared.
Meta-R	**Retrieve**, Retrieve the nth line currently being displayed in the scroller, go to EDIT mode, and insert the text up to the carriage return in the current line. If the first character of the retrieved line is a period, an asterisk or an SOS line number, those characters are not copied. If **n**<0, retrieve the nth line before the current line. If n=0 then a Control-U is sent to the PDP-10 and the current line is retrieved in edit mode. (Useful if you discover the line you are typing is in error.)

Keyboard Meta Characters

These commands control the keyboard functions. They are initialized with a Meta-Control-K.

Meta-L	**Local**, Place the terminal in local mode for non-Meta characters only. No characters are sent to the PDP-10.
Meta-N	**Normal**, Reset action for non-Meta characters to normal mode. Useful when exiting Local mode. Meta-N causes all non-Meta characters to be sent to the PDP-10.
Meta-C	**Clear**, Clear all the Meta characters of their actions. This is useful because each Meta-Control mode adds to the active Meta characters. Conflicting characters are replaced but old Meta characters with no equivalent in the new mode remain active.
Meta-W	**Space War**, Turn on Space War mode. In this mode, the keyboard interrupts twice for each key stroke, once when the key is depressed, and once when it is released.
Meta-P	**Peace**, Turn-off Space-War mode (default).
Meta-K	**Keyboard lock**, The keyboard number will be OR'ed with all future keyboard input. Appendix A shows the bit pattern received for a key stroke. With the proper value to OR in, this routine can cause future characters to be treated as though some of the special keys were depressed when the character was typed.
Meta-O	**Escape**, If the keyboard number is zero, turn off escape character sending. If non-zero (the default state) then send escape characters to the PDP-10 with each control character.

Intra-Line Editor

Normally, all non-Meta characters entered through the keyboard are sent immediately to the PDP-10. When the intra-line editor is enabled (with a Meta-Control-E), the line being typed is kept in a PDP-11 buffer so that editing may be done on that line before it is sent. At the conclusion of editing, the entire line is sent to the PDP-10 and another line may be entered.

The editor commands are modeled after the SOS intra-line edit commands, but provide easier editing in conjunction with the graphics system:

1) When editing is terminated (e.g. by typing a carriage return), the line is sent exactly as it then appears on the screen, so there can be no confusion as to the result of the editing.

2) The availability of Meta characters permits a clear distinction between text and editor commands.

The editor is turned on by typing Meta-Control-E. Text is then entered normally. The editor will maintain two cursors: the first cursor delimits text which has been sent to the PDP-10 from text that is being held in a PDP-11 buffer. The second cursor (the position-cursor) moves about in the PDP-11 buffer. Any text that is typed is entered in the buffer at the position-cursor location. Each text character typed causes the position-cursor to move one place to the right. At any time while entering text, the user may use the editing commands below to make corrections to what has been typed. When the line is complete, a break character (usually a carriage return) is typed, causing the line to be sent to the PDP-10 followed by the break character itself.

NOTE: The "break" characters are a subset of the control characters. Break characters cause transmission of the edit line followed by the character itself. Non-break control characters are sent immediately to the PDP-10 without affecting the edit line.

All editor commands except backspace (BS key) are single Meta characters. Some commands make use of the keyboard number, denoted below as **n**. "Text," when used below, refers to non-Meta, non-Control characters, i.e. what is usually to be entered into the line.

Meta-Space	Move right, Move cursor **n** characters to the right.
Meta-BS	Move left, Move cursor **n** characters to the left.
Meta-TAB	Move far right, Move cursor to the right end of the line.
Meta-RETURN	Move far left, Move cursor to the left end of the line.
Meta-I	Insert, Insert all following text characters at the present cursor position. Characters to the right of the cursor are pushed right to remain ahead

of the inserted text. This command is turned off by the next editor command. (NOTE: You are normally in over-write mode. Except after a Meta-I command, all entered text is entered at the cursor position, replacing any characters there previously.)

Meta-D	**Delete right,** Delete n characters right of the cursor. Text on the right side of the deleted characters is moved left.
BS	**Delete left,** Deletes n characters left of the position cursor. Text to the right of the cursor is moved to the left. Do not confuse this command with Meta-BS.
RETURN	**Done,** This is not really an editor command, but a break character. It will cause the entire line being edited (regardless of the position cursor location) to be sent to the PDP-10, followed by a RETURN character. A new line may then be entered and edited. Note that transmission occurs from the position of the left cursor. This is usually the beginning of the line, but not always (e.g. see the Meta-B command below).
Meta-S	**Skip,** This command causes the cursor to skip to the nth occurrence of the next character that you type. After the Meta-S, type one non-Meta character and the cursor will jump to the proper position.
Meta-K	**Kill,** This is the same as Meta-S except that all characters from the current cursor position up to the new cursor position are deleted.
Meta-E	**Exit,** Turn off the editor.
Meta-Q	**Quit,** Delete the buffer and sends a line-feed.
Meta-B	**Burst,** This command sends all characters up to the current cursor position back to the PDP-10. These characters are deleted from the buffer.
Meta-U	**Delete everything,** Delete all characters. Position cursor at left end of buffer.
Meta-)	**Delete all right,** Delete all characters to the right of the cursor.
Meta-(	**Delete all left,** Delete all characters to the left of the cursor.
Meta-LINE	**Retrieve,** This command retrieves the last edited line and puts it in the buffer for further editing.

References

Manuals about related systems:

GDP2.XGO[A630GS00]/B	Hardware documentation *
GLSTER.XGO[A630GS00]/B	Display package
GPI.XGO[A640SH01]/A	Convenient display package
SYS:LINK11.DOC	To link programs
SYS:DDT11.DOC	To debug programs
MONLOD.XGO[A630GS00]/B	Systems support
TABLET.XGO[A630GS00]	Tablet support
SILOS.XGO[A630KS00]/B	Character set editor
GDP.DEM[A630GS00]	Demonstration pointers
BSG.DOC[A630GS00]	SAIL-like runtime routines

Software aids:

SX.P11[A630GS00]	Macro-11 Trap defines
SX.B11[A630GS00]	BLISS-11 Trap defines
SX.SAI[A630GS00]	SAIL Trap defines
SX.BLI[A630GS00]	BLISS-10 Trap defines
GRADEF.P11[A630GS00]	Macro-11 Graphics defines
GRADEF.B11[A630GS00]	BLISS-11 Graphics defines
GRADEF.SAI[A630GS00]	SAIL Graphics defines
GRADEF.BLI[A630GS00]	BLISS-10 Graphics defines

* Available in document room as "GDP Programmers Guide"