UNIX^{TM} TIME-SHARING SYSTEM:

UNIX PROGRAMMER'S MANUAL

Seventh Edition, Volume 2A

January, 1979

Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey

UNIX Programmer's Manual
Volume 2 — Supplementary Documents

Seventh Edition

January 10, 1979

This volume contains documents which supplement the information contained in Volume 1 of *The UNIX*† *Programmer's Manual.* The documents here are grouped roughly into the areas of basics, editing, language tools, document preparation, and system maintenance. Further general information may be found in the Bell System Technical Journal special issue on UNIX, July-August, 1978.

Many of the documents cited within this volume as Bell Laboratories internal memoranda or Computing Science Technical Reports (CSTR) are also contained here.

These documents contain occasional localisms, typically references to other operating systems like GCOS and IBM. In all cases, such references may be safely ignored by UNIX users.

General Works

1. 7th Edition UNIX — Summary.
   A concise summary of the facilities available on UNIX.
2. The UNIX Time-Sharing System. D. M. Ritchie and K. Thompson.
   The original UNIX paper, reprinted from CACM.

Getting Started

3. UNIX for Beginners — Second Edition. B. W. Kernighan.
   An introduction to the most basic use of the system.
4. A Tutorial Introduction to the UNIX Text Editor. B. W. Kernighan.
   An easy way to get started with the editor.
5. Advanced Editing on UNIX. B. W. Kernighan.
   The next step.
6. An Introduction to the UNIX Shell. S. R. Bourne.
   An introduction to the capabilities of the command interpreter, the shell.
7. Learn — Computer Aided Instruction on UNIX. M. E. Lesk and B. W. Kernighan.
   Describes a computer-aided instruction program that walks new users through the basics of files, the editor, and document preparation software.

Document Preparation

8. Typing Documents on the UNIX System. M. E. Lesk.
   Describes the basic use of the formatting tools. Also describes ''−ms'', a standardized package of formatting requests that can be used to lay out most documents (including those in this volume).

---

9. A System for Typesetting Mathematics. B. W. Kernighan and L. L. Cherry.
   Describes EQN. an easy-to-learn language for doing high-quality mathematical typesetting,

10. TBL — A Program to Format Tables. M. E. Lesk.
    A program to permit easy specification of tabular material for typesetting. Again, easy to learn and use.

11. Some Applications of Inverted Indexes on the UNIX System. M. E. Lesk.
    Describes, among other things, the program REFER which fills in bibliographic citations from a data base automatically.

12. NROFF/TROFF User's Manual. J. F. Ossanna.
    The basic formatting program.

13. A TROFF Tutorial. B. W. Kernighan.
    An introduction to TROFF for those who really want to know such things.

Programming

14. The C Programming Language — Reference Manual. D. M. Ritchie.
    Official statement of the syntax and semantics of C. Should be supplemented by *The C Programming Language,* B. W. Kernighan and D. M. Ritchie, Prentice-Hall, 1978, which contains a tutorial introduction and many examples.

15. Lint, A C Program Checker. S. C. Johnson.
    Checks C programs for syntax errors, type violations, portability problems, and a variety of probable errors.

16. Make — A Program for Maintaining Computer Programs. S. I. Feldman.
    Indispensable tool for making sure that large programs are properly compiled with minimal effort.

17. UNIX Programming. B. W. Kernighan and D. M. Ritchie.
    Describes the programming interface to the operating system and the standard I/O library.

18. A Tutorial Introduction to ADB. J. F. Maranzano and S. R. Bourne.
    How to use the ADB debugger.

Supporting Tools and Languages

19. YACC: Yet Another Compiler-Compiler. S. C. Johnson.
    Converts a BNF specification of a language and semantic actions written in C into a compiler for the language.

20. LEX — A Lexical Analyzer Generator. M. E. Lesk and E. Schmidt.
    Creates a recognizer for a set of regular expressions; each regular expression can be followed by arbitrary C code which will be executed when the regular expression is found.

21. A Portable Fortran 77 Compiler. S. I. Feldman and P. J. Weinberger.
    The first Fortran 77 compiler, and still one of the best.

22. Ratfor — A Preprocessor for a Rational Fortran. B. W. Kernighan.
    Converts a Fortran with C-like control structures and cosmetics into real, ugly Fortran.

23. The M4 Macro Processor. B. W. Kernighan and D. M. Ritchie.
    M4 is a macro processor useful as a front end for C, Ratfor, Cobol, and in its own right.

24. SED — A Non-interactive Text Editor. L. E. McMahon.
    A variant of the editor for processing large inputs.

25. AWK — A Pattern Scanning and Processing Language. A. V. Aho, B. W. Kernighan and P. J. Weinberger.
    Makes it easy to specify many data transformation and selection operations.

26. DC — An Interactive Desk Calculator. R. H. Morris and L. L. Cherry.
    A super HP calculator, if you don't need floating point.

27. BC — An Arbitrary Precision Desk-Calculator Language. L. L. Cherry and R. H. Morris.
    A front end for DC that provides infix notation, control flow, and built-in functions.

28. UNIX Assembler Reference Manual. D. M. Ritchie.
    The ultimate dead language.

Implementation, Maintenance, and Miscellaneous

29. Setting Up UNIX — Seventh Edition. C. B. Haley and D. M. Ritchie.
    How to configure and get your system running.

30. Regenerating System Software. C. B. Haley and D. M. Ritchie.
    What do do when you have to change things.

31. UNIX Implementation. K. Thompson.
    How the system actually works inside.

32. The UNIX I/O System. D. M. Ritchie.
    How the I/O system really works.

33. A Tour Through the UNIX C Compiler. D. M. Ritchie.
    How the PDP-11 compiler works inside.

34. A Tour Through the Portable C Compiler. S. C. Johnson.
    How the portable C compiler works inside.

35. A Dial-Up Network of UNIX Systems. D. A. Nowitz and M. E. Lesk.
    Describes UUCP, a program for communicating files between UNIX systems.

36. UUCP Implementation Description. D. A. Nowitz.
    How UUCP works, and how to administer it.

37. On the Security of UNIX. D. M. Ritchie.
    Hints on how to break UNIX, and how to avoid doing so.

38. Password Security: A Case History. R. H. Morris and K. Thompson.
    How the bad guys used to be able to break the password algorithm, and why they can't now, at least not so easily.

7th Edition UNIX — Summary

September 6, 1978

Bell Laboratories
Murray Hill, New Jersey 07974

A. What's new: highlights of the 7th edition UNIX† System

Aimed at larger systems. Devices are addressable to 2^{31} bytes, files to 2^{30} bytes. 128K memory (separate instruction and data space) is needed for some utilities.

Portability. Code of the operating system and most utilities has been extensively revised to minimize its dependence on particular hardware.

Fortran 77. F77 compiler for the new standard language is compatible with C at the object level. A Fortran structurer, STRUCT, converts old, ugly Fortran into RATFOR, a structured dialect usable with F77.

Shell. Completely new SH program supports string variables, trap handling, structured programming, user profiles, settable search path, multilevel file name generation, etc.

Document preparation. TROFF phototypesetter utility is standard. NROFF (for terminals) is now highly compatible with TROFF. MS macro package provides canned commands for many common formatting and layout situations. TBL provides an easy to learn language for preparing complicated tabular material. REFER fills in bibliographic citations from a data base.

UNIX-to-UNIX file copy. UUCP performs spooled file transfers between any two machines.

Data processing. SED stream editor does multiple editing functions in parallel on a data stream of indefinite length. AWK report generator does free-field pattern selection and arithmetic operations.

Program development. MAKE controls re-creation of complicated software, arranging for minimal recompilation.

Debugging. ADB does postmortem and breakpoint debugging, handles separate instruction and data spaces, floating point, etc.

C language. The language now supports definable data types, generalized initialization, block structure, long integers, unions, explicit type conversions. The LINT verifier does strong type checking and detection of probable errors and portability problems even across separately compiled functions.

Lexical analyzer generator. LEX converts specification of regular expressions and semantic actions into a recognizing subroutine. Analogous to YACC.

Graphics. Simple graph-drawing utility, graphic subroutines, and generalized plotting filters adapted to various devices are now standard.

Standard input-output package. Highly efficient buffered stream I/O is integrated with formatted input and output.

Other. The operating system and utilities have been enhanced and freed of restrictions in many other ways too numerous to relate.

---

† UNIX is a Trademark of Bell Laboratories.

B. Hardware

The 7th edition UNIX operating system runs on a DEC PDP-11/45 or 11/70* with at least the following equipment:

128K to 2M words of managed memory; parity not used.

disk: RP03, RP04, RP06, RK05 (more than 1 RK05) or equivalent.

console typewriter.

clock: KW11-L or KW11-P.

The following equipment is strongly recommended:

communications controller such as DL11 or DH11.

full duplex 96-character ASCII terminals.

9-track tape or extra disk for system backup.

The system is normally distributed on 9-track tape. The minimum memory and disk space specified is enough to run and maintain UNIX. More will be needed to keep all source on line, or to handle a large number of users, big data bases, diversified complements of devices, or large programs. The resident code occupies 12-20K words depending on configuration; system data occupies 10-28K words.

There is no commitment to provide 7th edition UNIX on PDP-11/34, 11/40 and 11/60 hardware.

C. Software

Most of the programs available as UNIX commands are listed. Source code and printed manuals are distributed for all of the listed software except games. Almost all of the code is written in C. Commands are self-contained and do not require extra setup information, unless specifically noted as ‘‘interactive.’’ Interactive programs can be made to run from a prepared script simply by redirecting input. Most programs intended for interactive use (e.g., the editor) allow for an escape to command level (the Shell). Most file processing commands can also go from standard input to standard output (‘‘filters’’). The piping facility of the Shell may be used to connect such filters directly to the input or output of other programs.

1. Basic Software

This includes the time-sharing operating system with utilities, a machine language assembler and a compiler for the programming language C—enough software to write and run new applications and to maintain or modify UNIX itself.

1.1. Operating System

□ UNIX — The basic resident code on which everything else depends. Supports the system calls, and maintains the file system. A general description of UNIX design philosophy and system facilities appeared in the Communications of the ACM, July, 1974. A more extensive survey is in the Bell System Technical Journal for July-August 1978. Capabilities include:

- ○ Reentrant code for user processes.
- ○ Separate instruction and data spaces.
- ○ ‘‘Group’’ access permissions for cooperative projects, with overlapping memberships.
- ○ Alarm-clock timeouts.
- ○ Timer-interrupt sampling and interprocess monitoring for debugging and measurement.

---

*PDP is a Trademark of Digital Equipment Corporation.

○ Multiplexed I/O for machine-to-machine communication.

□ DEVICES All I/O is logically synchronous. I/O devices are simply files in the file system. Normally, invisible buffering makes all physical record structure and device characteristics transparent and exploits the hardware's ability to do overlapped I/O. Unbuffered physical record I/O is available for unusual applications. Drivers for these devices are available; others can be easily written:
○ Asynchronous interfaces: DH11, DL11. Support for most common ASCII terminals.
○ Synchronous interface: DP11.
○ Automatic calling unit interface: DN11.
○ Line printer: LP11.
○ Magnetic tape: TU10 and TU16.
○ DECtape: TC11.
○ Fixed head disk: RS11, RS03 and RS04.
○ Pack type disk: RP03, RP04, RP06; minimum-latency seek scheduling.
○ Cartridge-type disk: RK05, one or more physical devices per logical device.
○ Null device.
○ Physical memory of PDP-11, or mapped memory in resident system.
○ Phototypesetter: Graphic Systems System/1 through DR11C.

□ BOOT Procedures to get UNIX started.

□ MKCONF Tailor device-dependent system code to hardware configuration. As distributed, UNIX can be brought up directly on any acceptable CPU with any acceptable disk, any sufficient amount of core, and either clock. Other changes, such as optimal assignment of directories to devices, inclusion of floating point simulator, or installation of device names in file system, can then be made at leisure.

1.2. User Access Control

□ LOGIN Sign on as a new user.
○ Verify password and establish user's individual and group (project) identity.
○ Adapt to characteristics of terminal.
○ Establish working directory.
○ Announce presence of mail (from MAIL).
○ Publish message of the day.
○ Execute user-specified profile.
○ Start command interpreter or other initial program.

□ PASSWD Change a password.
○ User can change his own password.
○ Passwords are kept encrypted for security.

□ NEWGRP Change working group (project). Protects against unauthorized changes to projects.

1.3. Terminal Handling

□ TABS Set tab stops appropriately for specified terminal type.

□ STTY Set up options for optimal control of a terminal. In so far as they are deducible from the input, these options are set automatically by LOGIN.
○ Half vs. full duplex.
○ Carriage return+line feed vs. newline.
○ Interpretation of tabs.
○ Parity.

○ Mapping of upper case to lower.
○ Raw vs. edited input.
○ Delays for tabs, newlines and carriage returns.

1.4. File Manipulation

□ CAT — Concatenate one or more files onto standard output. Particularly used for unadorned printing, for inserting data into a pipeline, and for buffering output that comes in dribs and drabs. Works on any file regardless of contents.

□ CP — Copy one file to another, or a set of files to a directory. Works on any file regardless of contents.

□ PR — Print files with title, date, and page number on every page.
○ Multicolumn output.
○ Parallel column merge of several files.

□ LPR — Off-line print. Spools arbitrary files to the line printer.

□ CMP — Compare two files and report if different.

□ TAIL — Print last *n* lines of input
○ May print last *n* characters, or from *n* lines or characters to end.

□ SPLIT — Split a large file into more manageable pieces. Occasionally necessary for editing (ED).

□ DD — Physical file format translator, for exchanging data with foreign systems, especially IBM 370's.

□ SUM — Sum the words of a file.

1.5. Manipulation of Directories and File Names

□ RM — Remove a file. Only the name goes away if any other names are linked to the file.
○ Step through a directory deleting files interactively.
○ Delete entire directory hierarchies.

□ LN — ''Link'' another name (alias) to an existing file.

□ MV — Move a file or files. Used for renaming files.

□ CHMOD — Change permissions on one or more files. Executable by files' owner.

□ CHOWN — Change owner of one or more files.

□ CHGRP — Change group (project) to which a file belongs.

□ MKDIR — Make a new directory.

□ RMDIR — Remove a directory.

□ CD — Change working directory.

□ FIND — Prowl the directory hierarchy finding every file that meets specified criteria.
○ Criteria include:
name matches a given pattern,
creation date in given range,
date of last use in given range,
given permissions,
given owner,
given special file characteristics,
boolean combinations of above.

○ Any directory may be considered to be the root.
○ Perform specified command on each file found.

1.6. Running of Programs

□ SH — The Shell, or command language interpreter.
- ○ Supply arguments to and run any executable program.
- ○ Redirect standard input, standard output, and standard error files.
- ○ Pipes: simultaneous execution with output of one process connected to the input of another.
- ○ Compose compound commands using:
  - if ... then ... else,
  - case switches,
  - while loops,
  - for loops over lists,
  - break, continue and exit,
  - parentheses for grouping.
- ○ Initiate background processes.
- ○ Perform Shell programs, i.e., command scripts with substitutable arguments.
- ○ Construct argument lists from all file names satisfying specified patterns.
- ○ Take special action on traps and interrupts.
- ○ User-settable search path for finding commands.
- ○ Executes user-settable profile upon login.
- ○ Optionally announces presence of mail as it arrives.
- ○ Provides variables and parameters with default setting.

□ TEST — Tests for use in Shell conditionals.
- ○ String comparison.
- ○ File nature and accessibility.
- ○ Boolean combinations of the above.

□ EXPR — String computations for calculating command arguments.
- ○ Integer arithmetic
- ○ Pattern matching

□ WAIT — Wait for termination of asynchronously running processes.

□ READ — Read a line from terminal, for interactive Shell procedure.

□ ECHO — Print remainder of command line. Useful for diagnostics or prompts in Shell programs, or for inserting data into a pipeline.

□ SLEEP — Suspend execution for a specified time.

□ NOHUP — Run a command immune to hanging up the terminal.

□ NICE — Run a command in low (or high) priority.

□ KILL — Terminate named processes.

□ CRON — Schedule regular actions at specified times.
- ○ Actions are arbitrary programs.
- ○ Times are conjunctions of month, day of month, day of week, hour and minute. Ranges are specifiable for each.

□ AT — Schedule a one-shot action for an arbitrary time.

□ TEE — Pass data between processes and divert a copy into one or more files.

1.7. Status Inquiries

□ LS — List the names of one, several, or all files in one or more directories.
- ○ Alphabetic or temporal sorting, up or down.
- ○ Optional information: size, owner, group, date last modified, date last accessed, permissions, i-node number.

□ FILE — Try to determine what kind of information is in a file by consulting the file system index and by reading the file itself.

□ DATE — Print today's date and time. Has considerable knowledge of calendric and horological peculiarities.
- ○ May set UNIX's idea of date and time.

□ DF — Report amount of free space on file system devices.

□ DU — Print a summary of total space occupied by all files in a hierarchy.

□ QUOT — Print summary of file space usage by user id.

□ WHO — Tell who's on the system.
- ○ List of presently logged in users, ports and times on.
- ○ Optional history of all logins and logouts.

□ PS — Report on active processes.
- ○ List your own or everybody's processes.
- ○ Tell what commands are being executed.
- ○ Optional status information: state and scheduling info, priority, attached terminal, what it's waiting for, size.

□ IOSTAT — Print statistics about system I/O activity.

□ TTY — Print name of your terminal.

□ PWD — Print name of your working directory.

1.8. Backup and Maintenance

□ MOUNT — Attach a device containing a file system to the tree of directories. Protects against nonsense arrangements.

□ UMOUNT — Remove the file system contained on a device from the tree of directories. Protects against removing a busy device.

□ MKFS — Make a new file system on a device.

□ MKNOD — Make an i-node (file system entry) for a special file. Special files are physical devices, virtual devices, physical memory, etc.

□ TP

□ TAR — Manage file archives on magnetic tape or DECtape. TAR is newer.
- ○ Collect files into an archive.
- ○ Update DECtape archive by date.
- ○ Replace or delete DECtape files.
- ○ Print table of contents.
- ○ Retrieve from archive.

□ DUMP — Dump the file system stored on a specified device, selectively by date, or indiscriminately.

□ RESTOR Restore a dumped file system, or selectively retrieve parts thereof.

□ SU Temporarily become the super user with all the rights and privileges thereof. Requires a password.

□ DCHECK

□ ICHECK

□ NCHECK Check consistency of file system.
○ Print gross statistics: number of files, number of directories, number of special files, space used, space free.
○ Report duplicate use of space.
○ Retrieve lost space.
○ Report inaccessible files.
○ Check consistency of directories.
○ List names of all files.

□ CLRI Peremptorily expunge a file and its space from a file system. Used to repair damaged file systems.

□ SYNC Force all outstanding I/O on the system to completion. Used to shut down gracefully.

1.9. Accounting

The timing information on which the reports are based can be manually cleared or shut off completely.

□ AC Publish cumulative connect time report.
○ Connect time by user or by day.
○ For all users or for selected users.

□ SA Publish Shell accounting report. Gives usage information on each command executed.
○ Number of times used.
○ Total system time, user time and elapsed time.
○ Optional averages and percentages.
○ Sorting on various fields.

1.10. Communication

□ MAIL Mail a message to one or more users. Also used to read and dispose of incoming mail. The presence of mail is announced by LOGIN and optionally by SH.
○ Each message can be disposed of individually.
○ Messages can be saved in files or forwarded.

□ CALENDAR Automatic reminder service for events of today and tomorrow.

□ WRITE Establish direct terminal communication with another user.

□ WALL Write to all users.

□ MESG Inhibit receipt of messages from WRITE and WALL.

□ CU Call up another time-sharing system.
○ Transparent interface to remote machine.
○ File transmission.
○ Take remote input from local file or put remote output into local file.
○ Remote system need not be UNIX.

□ UUCP UNIX to UNIX copy.

○ Automatic queuing until line becomes available and remote machine is up.
○ Copy between two remote machines.
○ Differences, mail, etc., between two machines.

1.11. Basic Program Development Tools

Some of these utilities are used as integral parts of the higher level languages described in section 2.

□ AR — Maintain archives and libraries. Combines several files into one for housekeeping efficiency.
○ Create new archive.
○ Update archive by date.
○ Replace or delete files.
○ Print table of contents.
○ Retrieve from archive.

□ AS — Assembler. Similar to PAL-11, but different in detail.
○ Creates object program consisting of
  code, possibly read-only,
  initialized data or read-write code,
  uninitialized data.
○ Relocatable object code is directly executable without further transformation.
○ Object code normally includes a symbol table.
○ Multiple source files.
○ Local labels.
○ Conditional assembly.
○ ''Conditional jump'' instructions become branches or branches plus jumps depending on distance.

□ Library — The basic run-time library. These routines are used freely by all software.
○ Buffered character-by-character I/O.
○ Formatted input and output conversion (SCANF and PRINTF) for standard input and output, files, in-memory conversion.
○ Storage allocator.
○ Time conversions.
○ Number conversions.
○ Password encryption.
○ Quicksort.
○ Random number generator.
○ Mathematical function library, including trigonometric functions and inverses, exponential, logarithm, square root, bessel functions.

□ ADB — Interactive debugger.
○ Postmortem dumping.
○ Examination of arbitrary files, with no limit on size.
○ Interactive breakpoint debugging with the debugger as a separate process.
○ Symbolic reference to local and global variables.
○ Stack trace for C programs.
○ Output formats:
  1-, 2-, or 4-byte integers in octal, decimal, or hex
  single and double floating point
  character and string
  disassembled machine instructions
○ Patching.

○ Searching for integer, character, or floating patterns.
○ Handles separated instruction and data space.

□ OD — Dump any file. Output options include any combination of octal or decimal by words, octal by bytes, ASCII, opcodes, hexadecimal.
○ Range of dumping is controllable.

□ LD — Link edit. Combine relocatable object files. Insert required routines from specified libraries.
○ Resulting code may be sharable.
○ Resulting code may have separate instruction and data spaces.

□ LORDER — Places object file names in proper order for loading, so that files depending on others come after them.

□ NM — Print the namelist (symbol table) of an object program. Provides control over the style and order of names that are printed.

□ SIZE — Report the core requirements of one or more object files.

□ STRIP — Remove the relocation and symbol table information from an object file to save space.

□ TIME — Run a command and report timing information on it.

□ PROF — Construct a profile of time spent per routine from statistics gathered by time-sampling the execution of a program. Uses floating point.
○ Subroutine call frequency and average times for C programs.

□ MAKE — Controls creation of large programs. Uses a control file specifying source file dependencies to make new version; uses time last changed to deduce minimum amount of work necessary.
○ Knows about CC, YACC, LEX, etc.

1.12. UNIX Programmer's Manual

□ Manual — Machine-readable version of the UNIX Programmer's Manual.
○ System overview.
○ All commands.
○ All system calls.
○ All subroutines in C and assembler libraries.
○ All devices and other special files.
○ Formats of file system and kinds of files known to system software.
○ Boot and maintenance procedures.

□ MAN — Print specified manual section on your terminal.

1.13. Computer-Aided Instruction

□ LEARN — A program for interpreting CAI scripts, plus scripts for learning about UNIX by using it.
○ Scripts for basic files and commands, editor, advanced files and commands, EQN, MS macros, C programming language.

2. Languages

2.1. The C Language

□ CC — Compile and/or link edit programs in the C language. The UNIX operating system, most of the subsystems and C itself are written in C. For a full description of C, read *The C Programming Language,* Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1978.
- ○ General purpose language designed for structured programming.
- ○ Data types include character, integer, float, double, pointers to all types, functions returning above types, arrays of all types, structures and unions of all types.
- ○ Operations intended to give machine-independent control of full machine facility, including to-memory operations and pointer arithmetic.
- ○ Macro preprocessor for parameterized code and inclusion of standard files.
- ○ All procedures recursive, with parameters by value.
- ○ Machine-independent pointer manipulation.
- ○ Object code uses full addressing capability of the PDP-11.
- ○ Runtime library gives access to all system facilities.
- ○ Definable data types.
- ○ Block structure

□ LINT — Verifier for C programs. Reports questionable or nonportable usage such as:

   Mismatched data declarations and procedure interfaces.
   Nonportable type conversions.
   Unused variables, unreachable code, no-effect operations.
   Mistyped pointers.
   Obsolete syntax.
- ○ Full cross-module checking of separately compiled programs.

□ CB — A beautifier for C programs. Does proper indentation and placement of braces.

2.2. Fortran

□ F77 — A full compiler for ANSI Standard Fortran 77.
- ○ Compatible with C and supporting tools at object level.
- ○ Optional source compatibility with Fortran 66.
- ○ Free format source.
- ○ Optional subscript-range checking, detection of uninitialized variables.
- ○ All widths of arithmetic: 2- and 4-byte integer; 4- and 8-byte real; 8- and 16-byte complex.

□ RATFOR — Ratfor adds rational control structure à la C to Fortran.
- ○ Compound statements.
- ○ If-else, do, for, while, repeat-until, break, next statements.
- ○ Symbolic constants.
- ○ File insertion.
- ○ Free format source
- ○ Translation of relationals like >, >=.
- ○ Produces genuine Fortran to carry away.
- ○ May be used with F77.

□ STRUCT — Converts ordinary ugly Fortran into structured Fortran (i.e., Ratfor), using statement grouping, if-else, while, for, repeat-until.

2.3. Other Algorithmic Languages

□ BAS — An interactive interpreter, similar in style to BASIC. Interpret unnumbered statements immediately, numbered statements upon ‘run’.

○ Statements include:
  comment,
  dump,
  for...next,
  goto,
  if...else...fi,
  list,
  print,
  prompt,
  return,
  run,
  save.
○ All calculations double precision.
○ Recursive function defining and calling.
○ Builtin functions include log, exp, sin, cos, atn, int, sqr, abs, rnd.
○ Escape to ED for complex program editing.

□ DC — Interactive programmable desk calculator. Has named storage locations as well as conventional stack for holding integers or programs.
○ Unlimited precision decimal arithmetic.
○ Appropriate treatment of decimal fractions.
○ Arbitrary input and output radices, in particular binary, octal, decimal and hexadecimal.
○ Reverse Polish operators:
  + − * /
  remainder, power, square root,
  load, store, duplicate, clear,
  print, enter program text, execute.

□ BC — A C-like interactive interface to the desk calculator DC.
○ All the capabilities of DC with a high-level syntax.
○ Arrays and recursive functions.
○ Immediate evaluation of expressions and evaluation of functions upon call.
○ Arbitrary precision elementary functions: exp, sin, cos, atan.
○ Go-to-less programming.

2.4. Macroprocessing

□ M4 — A general purpose macroprocessor.
○ Stream-oriented, recognizes macros anywhere in text.
○ Syntax fits with functional syntax of most higher-level languages.
○ Can evaluate integer arithmetic expressions.

2.5. Compiler-compilers

□ YACC — An LR(1)-based compiler writing system. During execution of resulting parsers, arbitrary C functions may be called to do code generation or semantic actions.
○ BNF syntax specifications.
○ Precedence relations.
○ Accepts formally ambiguous grammars with non-BNF resolution rules.

□ LEX — Generator of lexical analyzers. Arbitrary C functions may be called upon isolation of each lexical token.

○ Full regular expression, plus left and right context dependence.
○ Resulting lexical analysers interface cleanly with YACC parsers.

3. Text Processing

3.1. Document Preparation

□ ED — Interactive context editor. Random access to all lines of a file.
- ○ Find lines by number or pattern. Patterns may include: specified characters, don't care characters, choices among characters, repetitions of these constructs, beginning of line, end of line.
- ○ Add, delete, change, copy, move or join lines.
- ○ Permute or split contents of a line.
- ○ Replace one or all instances of a pattern within a line.
- ○ Combine or split files.
- ○ Escape to Shell (command language) during editing.
- ○ Do any of above operations on every pattern-selected line in a given range.
- ○ Optional encryption for extra security.

□ PTX — Make a permuted (key word in context) index.

□ SPELL — Look for spelling errors by comparing each word in a document against a word list.
- ○ 25,000-word list includes proper names.
- ○ Handles common prefixes and suffixes.
- ○ Collects words to help tailor local spelling lists.

□ LOOK — Search for words in dictionary that begin with specified prefix.

□ TYPO — Look for spelling errors by a statistical technique; not limited to English.

□ CRYPT — Encrypt and decrypt files for security.

3.2. Document Formatting

□ ROFF — A typesetting program for terminals. Easy for nontechnical people to learn, and good for simple documents. Input consists of data lines intermixed with control lines, such as

ROFF is deemed to be obsolete;
it is intended only for casual use.
- ○ Justification of either or both margins.
- ○ Automatic hyphenation.
- ○ Generalized running heads and feet, with even-odd page capability, numbering, etc.
- ○ Definable macros for frequently used control sequences (no substitutable arguments).
- ○ All 4 margins and page size dynamically adjustable.
- ○ Hanging indents and one-line indents.
- ○ Absolute and relative parameter settings.
- ○ Optional legal-style numbering of output lines.
- ○ Multiple file capability.
- ○ Not usable as a filter.

□ TROFF

□ NROFF — Advanced typesetting. TROFF drives a Graphic Systems phototypesetter; NROFF drives ascii terminals of all types. This summary was typeset using TROFF. TROFF and NROFF style is similar to ROFF, but they are capable of much more elaborate feats of formatting, when appropriately programmed. TROFF and NROFF accept the

same input language.
○ All ROFF capabilities available or definable.
○ Completely definable page format keyed to dynamically planted ‘‘interrupts’’ at specified lines.
○ Maintains several separately definable typesetting environments (e.g., one for body text, one for footnotes, and one for unusually elaborate headings).
○ Arbitrary number of output pools can be combined at will.
○ Macros with substitutable arguments, and macros invocable in mid-line.
○ Computation and printing of numerical quantities.
○ Conditional execution of macros.
○ Tabular layout facility.
○ Positions expressible in inches, centimeters, ems, points, machine units or arithmetic combinations thereof.
○ Access to character-width computation for unusually difficult layout problems.
○ Overstrikes, built-up brackets, horizontal and vertical line drawing.
○ Dynamic relative or absolute positioning and size selection, globally or at the character level.
○ Can exploit the characteristics of the terminal being used, for approximating special characters, reverse motions, proportional spacing, etc.

The Graphic Systems typesetter has a vocabulary of several 102-character fonts (4 simultaneously) in 15 sizes. TROFF provides terminal output for rough sampling of the product.

NROFF will produce multicolumn output on terminals capable of reverse line feed, or through the post-processor COL.

High programming skill is required to exploit the formatting capabilities of TROFF and NROFF, although unskilled personnel can easily be trained to enter documents according to canned formats such as those provided by MS, below. TROFF and EQN are essentially identical to NROFF and NEQN so it is usually possible to define interchangeable formats to produce approximate proof copy on terminals before actual typesetting. The preprocessors MS, TBL, and REFER are fully compatible with TROFF and NROFF.

□ MS — A standardized manuscript layout package for use with NROFF/TROFF. This document was formatted with MS.
○ Page numbers and draft dates.
○ Automatically numbered subheads.
○ Footnotes.
○ Single or double column.
○ Paragraphing, display and indentation.
○ Numbered equations.

□ EQN — A mathematical typesetting preprocessor for TROFF. Translates easily readable formulas, either in-line or displayed, into detailed typesetting instructions. Formulas are written in a style like this:

```
sigma sup 2 ~=~ 1 over N sum from i=1 to N ( x sub i − x bar ) sup 2
```

which produces:

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})^2$$

○ Automatic calculation of size changes for subscripts, sub-subscripts, etc.
○ Full vocabulary of Greek letters and special symbols, such as ‘gamma’, ‘GAMMA’, ‘integral’.

○ Automatic calculation of large bracket sizes.
○ Vertical ‘‘piling’’ of formulae for matrices, conditional alternatives, etc.
○ Integrals, sums, etc., with arbitrarily complex limits.
○ Diacriticals: dots, double dots, hats, bars, etc.
○ Easily learned by nonprogrammers and mathematical typists.

□ NEQN — A version of EQN for NROFF; accepts the same input language. Prepares formulas for display on any terminal that NROFF knows about, for example, those based on Diablo printing mechanism.
○ Same facilities as EQN within graphical capability of terminal.

□ TBL — A preprocessor for NROFF/TROFF that translates simple descriptions of table layouts and contents into detailed typesetting instructions.
○ Computes column widths.
○ Handles left- and right-justified columns, centered columns and decimal-point alignment.
○ Places column titles.
○ Table entries can be text, which is adjusted to fit.
○ Can box all or parts of table.

□ REFER — Fills in bibliographic citations in a document from a data base (not supplied).
○ References may be printed in any style, as they occur or collected at the end.
○ May be numbered sequentially, by name of author, etc.

□ TC — Simulate Graphic Systems typesetter on Tektronix 4014 scope. Useful for checking TROFF page layout before typesetting.

□ GREEK — Fancy printing on Diablo-mechanism terminals like DASI-300 and DASI-450, and on Tektronix 4014.
○ Gives half-line forward and reverse motions.
○ Approximates Greek letters and other special characters by overstriking.

□ COL — Canonicalize files with reverse line feeds for one-pass printing.

□ DEROFF — Remove all TROFF commands from input.

□ CHECKEQ — Check document for possible errors in EQN usage.

4. Information Handling

□ SORT — Sort or merge ASCII files line-by-line. No limit on input size.
○ Sort up or down.
○ Sort lexicographically or on numeric key.
○ Multiple keys located by delimiters or by character position.
○ May sort upper case together with lower into dictionary order.
○ Optionally suppress duplicate data.

□ TSORT — Topological sort — converts a partial order into a total order.

□ UNIQ — Collapse successive duplicate lines in a file into one line.
○ Publish lines that were originally unique, duplicated, or both.
○ May give redundancy count for each line.

□ TR — Do one-to-one character translation according to an arbitrary code.
○ May coalesce selected repeated characters.
○ May delete selected characters.

□ DIFF — Report line changes, additions and deletions necessary to bring two files into agreement.

○ May produce an editor script to convert one file into another.
○ A variant compares two new versions against one old one.

□ COMM Identify common lines in two sorted files. Output in up to 3 columns shows lines present in first file only, present in both, and/or present in second only.

□ JOIN Combine two files by joining records that have identical keys.

□ GREP Print all lines in a file that satisfy a pattern as used in the editor ED.
○ May print all lines that fail to match.
○ May print count of hits.
○ May print first hit in each file.

□ LOOK Binary search in sorted file for lines with specified prefix.

□ WC Count the lines, ‘‘words’’ (blank-separated strings) and characters in a file.

□ SED Stream-oriented version of ED. Can perform a sequence of editing operations on each line of an input stream of unbounded length.
○ Lines may be selected by address or range of addresses.
○ Control flow and conditional testing.
○ Multiple output streams.
○ Multi-line capability.

□ AWK Pattern scanning and processing language. Searches input for patterns, and performs actions on each line of input that satisfies the pattern.
○ Patterns include regular expressions, arithmetic and lexicographic conditions, boolean combinations and ranges of these.
○ Data treated as string or numeric as appropriate.
○ Can break input into fields; fields are variables.
○ Variables and arrays (with non-numeric subscripts).
○ Full set of arithmetic operators and control flow.
○ Multiple output streams to files and pipes.
○ Output can be formatted as desired.
○ Multi-line capabilities.

5. Graphics

The programs in this section are predominantly intended for use with Tektronix 4014 storage scopes.

□ GRAPH Prepares a graph of a set of input numbers.
○ Input scaled to fit standard plotting area.
○ Abscissae may be supplied automatically.
○ Graph may be labeled.
○ Control over grid style, line style, graph orientation, etc.

□ SPLINE Provides a smooth curve through a set of points intended for GRAPH.

□ PLOT A set of filters for printing graphs produced by GRAPH and other programs on various terminals. Filters provided for 4014, DASI terminals, Versatec printer/plotter.

6. Novelties, Games, and Things That Didn’t Fit Anywhere Else

□ BACKGAMMON
A player of modest accomplishment.

□ CHESS Plays good class D chess.

□ CHECKERS Ditto, for checkers.

□ BCD Converts ascii to card-image form.

□ PPT Converts ascii to paper tape form.

□ BJ A blackjack dealer.

□ CUBIC An accomplished player of 4×4×4 tic-tac-toe.

□ MAZE Constructs random mazes for you to solve.

□ MOO A fascinating number-guessing game.

□ CAL Print a calendar of specified month and year.

□ BANNER Print output in huge letters.

□ CHING The *I Ching*. Place your own interpretation on the output.

□ FORTUNE Presents a random fortune cookie on each invocation. Limited jar of cookies included.

□ UNITS Convert amounts between different scales of measurement. Knows hundreds of units. For example, how many km/sec is a parsec/megayear?

□ TTT A tic-tac-toe program that learns. It never makes the same mistake twice.

□ ARITHMETIC
Speed and accuracy test for number facts.

□ FACTOR Factor large integers.

□ QUIZ Test your knowledge of Shakespeare, Presidents, capitals, etc.

□ WUMP Hunt the wumpus, thrilling search in a dangerous cave.

□ REVERSI A two person board game, isomorphic to Othello®.

□ HANGMAN Word-guessing game. Uses the dictionary supplied with SPELL.

□ FISH Children’s card-guessing game.

The UNIX Time-Sharing System*

D. M. Ritchie and K. Thompson

ABSTRACT

UNIX† is a general-purpose, multi-user, interactive operating system for the larger Digital Equipment Corporation PDP-11 and the Interdata 8/32 computers. It offers a number of features seldom found even in larger operating systems, including

i A hierarchical file system incorporating demountable volumes,

ii Compatible file, device, and inter-process I/O,

iii The ability to initiate asynchronous processes,

iv System command language selectable on a per-user basis,

v Over 100 subsystems including a dozen languages,

vi High degree of portability.

This paper discusses the nature and implementation of the file system and of the user command interface.

1. INTRODUCTION

There have been four versions of the UNIX time-sharing system. The earliest (circa 1969-70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. The third incorporated multiprogramming and ran on the PDP-11/34, /40, /45, /60, and /70 computers; it is the one described in the previously published version of this paper, and is also the most widely used today. This paper describes only the fourth, current system that runs on the PDP-11/70 and the Interdata 8/32 computers. In fact, the differences among the various systems is rather small; most of the revisions made to the originally published version of this paper, aside from those concerned with style, had to do with details of the implementation of the file system.

Since PDP-11 UNIX became operational in February, 1971, over 600 installations have been put into service. Most of them are engaged in applications such as computer science education, the preparation and formatting of documents and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: it can run on hardware costing as little as $40,000, and less than two man-years were spent on the main system software. We hope, however, that users find that the most important characteristics of the system are its simplicity, elegance, and ease of use.

Besides the operating system proper, some major programs available under UNIX are

---

 This is a revised version of an article that appeared in Communications of the ACM, *17*, No. 7 (July 1974), pp. 365-375. That article was a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973.

C compiler
Text editor based on QED[1]
Assembler, linking loader, symbolic debugger
Phototypesetting and equation setting programs[2,3]
Dozens of languages including Fortran 77, Basic, Snobol, APL, Algol 68, M6, TMG, Pascal

There is a host of maintenance, utility, recreation and novelty programs, all written locally. The UNIX user community, which numbers in the thousands, has contributed many more programs and languages. It is worth noting that the system is totally self-supporting. All UNIX software is maintained on the system; likewise, this paper and all other documents in this issue were generated and formatted by the UNIX editor and text formatting programs.

II. HARDWARE AND SOFTWARE ENVIRONMENT

The PDP-11/70 on which the Research UNIX system is installed is a 16-bit word (8-bit byte) computer with 768K bytes of core memory; the system kernel occupies 90K bytes about equally divided between code and data tables. This system, however, includes a very large number of device drivers and enjoys a generous allotment of space for I/O buffers and system tables; a minimal system capable of running the software mentioned above can require as little as 96K bytes of core altogether. There are even larger installations; see the description of the PWB/UNIX systems,[4,5] for example. There are also much smaller, though somewhat restricted, versions of the system.[6]

Our own PDP-11 has two 200-Mb moving-head disks for file system storage and swapping. There are 20 variable-speed communications interfaces attached to 300- and 1200-baud data sets, and an additional 12 communication lines hard-wired to 9600-baud terminals and satellite computers. There are also several 2400- and 4800-baud synchronous communication interfaces used for machine-to-machine file transfer. Finally, there is a variety of miscellaneous devices including nine-track magnetic tape, a line printer, a voice synthesizer, a phototypesetter, a digital switching network, and a chess machine.

The preponderance of UNIX software is written in the abovementioned C language.[7] Early versions of the operating system were written in assembly language, but during the summer of 1973, it was rewritten in C. The size of the new system was about one-third greater than that of the old. Since the new system not only became much easier to understand and to modify but also included many functional improvements, including multiprogramming and the ability to share reentrant code among several user programs, we consider this increase in size quite acceptable.

III. THE FILE SYSTEM

The most important role of the system is to provide a file system. From the point of view of the user, there are three kinds of files: ordinary disk files, directories, and special files.

3.1 Ordinary files

A file contains whatever information the user places on it, for example, symbolic or binary (object) programs. No particular structuring is expected by the system. A file of text consists simply of a string of characters, with lines demarcated by the newline character. Binary programs are sequences of words as they will appear in core memory when the program starts executing. A few user programs manipulate files with more structure; for example, the assembler generates, and the loader expects, an object file in a particular format. However, the structure of files is controlled by the programs that use them, not by the system.

3.2 Directories

Directories provide the mapping between the names of files and the files themselves, and thus induce a structure on the file system as a whole. Each user has a directory of his own files; he may also create subdirectories to contain groups of files conveniently treated together. A directory behaves exactly like an ordinary file except that it cannot be written on by unprivileged programs, so that the system controls the contents of directories. However, anyone with appropriate permission may read a directory just like any other file.

The system maintains several directories for its own use. One of these is the **root** directory. All files in the system can be found by tracing a path through a chain of directories until the desired file is reached. The starting point for such searches is often the **root**. Other system directories contain all the programs provided for general use; that is, all the *commands*. As will be seen, however, it is by no means necessary that a program reside in one of these directories for it to be executed.

Files are named by sequences of 14 or fewer characters. When the name of a file is specified to the system, it may be in the form of a *path name*, which is a sequence of directory names separated by slashes, ‘‘/’’, and ending in a file name. If the sequence begins with a slash, the search begins in the root directory. The name **/alpha/beta/gamma** causes the system to search the root for directory **alpha**, then to search **alpha** for **beta**, finally to find **gamma** in **beta**. **gamma** may be an ordinary file, a directory, or a special file. As a limiting case, the name ‘‘/’’ refers to the root itself.

A path name not starting with ‘‘/’’ causes the system to begin the search in the user’s current directory. Thus, the name **alpha/beta** specifies the file named **beta** in subdirectory **alpha** of the current directory. The simplest kind of name, for example, **alpha**, refers to a file that itself is found in the current directory. As another limiting case, the null file name refers to the current directory.

The same non-directory file may appear in several directories under possibly different names. This feature is called *linking*; a directory entry for a file is sometimes called a link. The UNIX system differs from other systems in which linking is permitted in that all links to a file have equal status. That is, a file does not exist within a particular directory; the directory entry for a file consists merely of its name and a pointer to the information actually describing the file. Thus a file exists independently of any directory entry, although in practice a file is made to disappear along with the last link to it.

Each directory always has at least two entries. The name ‘‘**.**’’ in each directory refers to the directory itself. Thus a program may read the current directory under the name ‘‘**.**’’ without knowing its complete path name. The name ‘‘**..**’’ by convention refers to the parent of the directory in which it appears, that is, to the directory in which it was created.

The directory structure is constrained to have the form of a rooted tree. Except for the special entries ‘‘**.**’’ and ‘‘**..**’’, each directory must appear as an entry in exactly one other directory, which is its parent. The reason for this is to simplify the writing of programs that visit subtrees of the directory structure, and more important, to avoid the separation of portions of the hierarchy. If arbitrary links to directories were permitted, it would be quite difficult to detect when the last connection from the root to a directory was severed.

3.3 Special files

Special files constitute the most unusual feature of the UNIX file system. Each supported I/O device is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory **/dev**, although a link may be made to one of these files just as it may to an ordinary file. Thus, for example, to write on a magnetic tape one may write on the file **/dev/mt**. Special files exist for each communication line, each disk, each tape drive, and for physical main memory. Of course, the active disks and the memory special file are protected from indiscriminate access.

There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same protection mechanism as regular files.

3.4 Removable file systems

Although the root of the file system is always stored on the same device, it is not necessary that the entire file system hierarchy reside on this device. There is a **mount** system request with two arguments: the name of an existing ordinary file, and the name of a special file whose associated storage volume (e.g., a disk pack) should have the structure of an independent file system containing its own directory hierarchy. The effect of **mount** is to cause references to the heretofore ordinary file to refer instead to the root directory of the file system on the removable volume. In effect, **mount** replaces a leaf of the hierarchy tree (the ordinary file) by a whole new subtree (the hierarchy stored on the removable volume). After the **mount**, there is virtually no distinction between files on the removable volume and those in the permanent file system. In our installation, for example, the root directory resides on a small partition of one of our disk drives, while the other drive, which contains the user's files, is mounted by the system initialization sequence. A mountable file system is generated by writing on its corresponding special file. A utility program is available to create an empty file system, or one may simply copy an existing file system.

There is only one exception to the rule of identical treatment of files on different devices: no link may exist between one file system hierarchy and another. This restriction is enforced so as to avoid the elaborate bookkeeping that would otherwise be required to assure removal of the links whenever the removable volume is dismounted.

3.5 Protection

Although the access control scheme is quite simple, it has some unusual features. Each user of the system is assigned a unique user identification number. When a file is created, it is marked with the user ID of its owner. Also given for new files is a set of ten protection bits. Nine of these specify independently read, write, and execute permission for the owner of the file, for other members of his group, and for all remaining users.

If the tenth bit is on, the system will temporarily change the user identification (hereafter, user ID) of the current user to that of the creator of the file whenever the file is executed as a program. This change in user ID is effective only during the execution of the program that calls for it. The set-user-ID feature provides for privileged programs that may use files inaccessible to other users. For example, a program may keep an accounting file that should neither be read nor changed except by the program itself. If the set-user-ID bit is on for the program, it may access the file although this access might be forbidden to other programs invoked by the given program's user. Since the actual user ID of the invoker of any program is always available, set-user-ID programs may take any measures desired to satisfy themselves as to their invoker's credentials. This mechanism is used to allow users to execute the carefully written commands that call privileged system entries. For example, there is a system entry invokable only by the ''super-user'' (below) that creates an empty directory. As indicated above, directories are expected to have entries for ''.'' and ''..''. The command which creates a directory is owned by the super-user and has the set-user-ID bit set. After it checks its invoker's authorization to create the specified directory, it creates it and makes the entries for ''.'' and ''..''.

Because anyone may set the set-user-ID bit on one of his own files, this mechanism is generally available without administrative intervention. For example, this protection scheme easily solves the MOO accounting problem posed by ''Aleph-null.''[8]

The system recognizes one particular user ID (that of the ''super-user'') as exempt from the usual constraints on file access; thus (for example), programs may be written to dump and reload the file system without unwanted interference from the protection system.

3.6 I/O calls

The system calls to do I/O are designed to eliminate the differences between the various devices and styles of access. There is no distinction between ''random'' and ''sequential'' I/O, nor is any logical record size imposed by the system. The size of an ordinary file is determined by the number of bytes written on it; no predetermination of the size of a file is necessary or possible.

To illustrate the essentials of I/O, some of the basic calls are summarized below in an anonymous language that will indicate the required parameters without getting into the underlying complexities. Each call to the system may potentially result in an error return, which for simplicity is not represented in the calling sequence.

To read or write a file assumed to exist already, it must be opened by the following call:

filep = open (name, flag)

where **name** indicates the name of the file. An arbitrary path name may be given. The **flag** argument indicates whether the file is to be read, written, or ''updated,'' that is, read and written simultaneously.

The returned value **filep** is called a *file descriptor*. It is a small integer used to identify the file in subsequent calls to read, write, or otherwise manipulate the file.

To create a new file or completely rewrite an old one, there is a **create** system call that creates the given file if it does not exist, or truncates it to zero length if it does exist; **create** also opens the new file for writing and, like **open**, returns a file descriptor.

The file system maintains no locks visible to the user, nor is there any restriction on the number of users who may have a file open for reading or writing. Although it is possible for the contents of a file to become scrambled when two users write on it simultaneously, in practice difficulties do not arise. We take the view that locks are neither necessary nor sufficient, in our environment, to prevent interference between users of the same file. They are unnecessary because we are not faced with large, single-file data bases maintained by independent processes. They are insufficient because locks in the ordinary sense, whereby one user is prevented from writing on a file that another user is reading, cannot prevent confusion when, for example, both users are editing a file with an editor that makes a copy of the file being edited.

There are, however, sufficient internal interlocks to maintain the logical consistency of the file system when two users engage simultaneously in activities such as writing on the same file, creating files in the same directory, or deleting each other's open files.

Except as indicated below, reading and writing are sequential. This means that if a particular byte in the file was the last byte written (or read), the next I/O call implicitly refers to the immediately following byte. For each open file there is a pointer, maintained inside the system, that indicates the next byte to be read or written. If *n* bytes are read or written, the pointer advances by *n* bytes.

Once a file is open, the following calls may be used:

n = read (filep, buffer, count)
n = write (filep, buffer, count)

Up to **count** bytes are transmitted between the file specified by **filep** and the byte array specified by **buffer**. The returned value **n** is the number of bytes actually transmitted. In the **write** case, **n** is the same as **count** except under exceptional conditions, such as I/O errors or end of physical medium on special files; in a **read**, however, **n** may without error be less than **count**. If the read pointer is so near the end of the file that reading **count** characters would cause reading beyond the end, only sufficient bytes are transmitted to reach the end of the file; also, typewriter-like terminals never return more than one line of input. When a **read** call returns with **n** equal to zero, the end of the file has been reached. For disk files this occurs when the read pointer becomes equal to the current size of the file. It is possible to generate an end-of-file from a terminal by use of an escape sequence that depends on the device used.

Bytes written affect only those parts of a file implied by the position of the write pointer and the count; no other part of the file is changed. If the last byte lies beyond the end of the file, the file is made to grow as needed.

To do random (direct-access) I/O it is only necessary to move the read or write pointer to the appropriate location in the file.

location = lseek (filep, offset, base)

The pointer associated with **filep** is moved to a position **offset** bytes from the beginning of the file, from the current position of the pointer, or from the end of the file, depending on **base. offset** may be negative. For some devices (e.g., paper tape and terminals) seek calls are ignored. The actual offset from the beginning of the file to which the pointer was moved is returned in **location**.

There are several additional system entries having to do with I/O and with the file system that will not be discussed. For example: close a file, get the status of a file, change the protection mode or the owner of a file, create a directory, make a link to an existing file, delete a file.

IV. IMPLEMENTATION OF THE FILE SYSTEM

As mentioned in Section 3.2 above, a directory entry contains only a name for the associated file and a pointer to the file itself. This pointer is an integer called the *i-number* (for index number) of the file. When the file is accessed, its i-number is used as an index into a system table (the *i-list*) stored in a known part of the device on which the directory resides. The entry found thereby (the file's *i-node*) contains the description of the file:

i the user and group-ID of its owner

ii its protection bits

iii the physical disk or tape addresses for the file contents

iv its size

v time of creation, last use, and last modification

vi the number of links to the file, that is, the number of times it appears in a directory

vii a code indicating whether the file is a directory, an ordinary file, or a special file.

The purpose of an **open** or **create** system call is to turn the path name given by the user into an i-number by searching the explicitly or implicitly named directories. Once a file is open, its device, i-number, and read/write pointer are stored in a system table indexed by the file descriptor returned by the **open** or **create**. Thus, during a subsequent call to read or write the file, the descriptor may be easily related to the information necessary to access the file.

When a new file is created, an i-node is allocated for it and a directory entry is made that contains the name of the file and the i-node number. Making a link to an existing file involves creating a directory entry with the new name, copying the i-number from the original file entry, and incrementing the link-count field of the i-node. Removing (deleting) a file is done by decrementing the link-count of the i-node specified by its directory entry and erasing the directory entry. If the link-count drops to 0, any disk blocks in the file are freed and the i-node is de-allocated.

The space on all disks that contain a file system is divided into a number of 512-byte blocks logically addressed from 0 up to a limit that depends on the device. There is space in the i-node of each file for 13 device addresses. For nonspecial files, the first 10 device addresses point at the first 10 blocks of the file. If the file is larger than 10 blocks, the 11 device address points to an indirect block containing up to 128 addresses of additional blocks in the file. Still larger files use the twelfth device address of the i-node to point to a double-indirect block naming 128 indirect blocks, each pointing to 128 blocks of the file. If required, the thirteenth device address is a triple-indirect block. Thus files may conceptually grow to $[(10+128+128^2+128^3)\cdot 512]$ bytes. Once opened, bytes numbered below 5120 can be read with a single disk access; bytes in the range 5120 to 70,656 require two accesses; bytes in the range 70,656 to 8,459,264 require three accesses; bytes from there to the largest file (1,082,201,088) require four accesses. In practice, a device cache mechanism (see below) proves effective in eliminating most of the indirect fetches.

The foregoing discussion applies to ordinary files. When an I/O request is made to a file whose i-node indicates that it is special, the last 12 device address words are immaterial, and the first specifies an internal *device name*, which is interpreted as a pair of numbers representing, respectively, a device type and subdevice number. The device type indicates which system routine will deal with I/O on that device; the subdevice number selects, for example, a disk drive attached to a particular controller or one of several similar terminal interfaces.

In this environment, the implementation of the **mount** system call (Section 3.4) is quite straightforward. **mount** maintains a system table whose argument is the i-number and device name of the ordinary file specified during the **mount**, and whose corresponding value is the device name of the indicated special file. This table is searched for each i-number/device pair that turns up while a path name is being scanned during an **open** or **create**; if a match is found, the i-number is replaced by the i-number of the root directory and the device name is replaced by the table value.

To the user, both reading and writing of files appear to be synchronous and unbuffered. That is, immediately after return from a **read** call the data are available; conversely, after a **write** the user's workspace may be reused. In fact, the system maintains a rather complicated buffering mechanism that reduces greatly the number of I/O operations required to access a file. Suppose a **write** call is made specifying transmission of a single byte. The system will search its buffers to see whether the affected disk block currently resides in main memory; if not, it will be read in from the device. Then the affected byte is replaced in the buffer and an entry is made in a list of blocks to be written. The return from the **write** call may then take place, although the actual I/O may not be completed until a later time. Conversely, if a single byte is read, the system determines whether the secondary storage block in which the byte is located is already in one of the system's buffers; if so, the byte can be returned immediately. If not, the block is read into a buffer and the byte picked out.

The system recognizes when a program has made accesses to sequential blocks of a file, and asynchronously pre-reads the next block. This significantly reduces the running time of most programs while adding little to system overhead.

A program that reads or writes files in units of 512 bytes has an advantage over a program that reads or writes a single byte at a time, but the gain is not immense; it comes mainly from the avoidance of system overhead. If a program is used rarely or does no great volume of I/O, it may quite reasonably read and write in units as small as it wishes.

The notion of the i-list is an unusual feature of UNIX. In practice, this method of organizing the file system has proved quite reliable and easy to deal with. To the system itself, one of its strengths is the fact that each file has a short, unambiguous name related in a simple way to the protection, addressing, and other information needed to access the file. It also permits a quite simple and rapid algorithm for checking the consistency of a file system, for example, verification that the portions of each device containing useful information and those free to be allocated are disjoint and together exhaust the space on the device. This algorithm is independent of the directory hierarchy, because it need only scan the linearly organized i-list. At the same time the notion of the i-list induces certain peculiarities not found in other file system organizations. For example, there is the question of who is to be charged for the space a file occupies, because all directory entries for a file have equal status. Charging the owner of a file is unfair in general, for one user may create a file, another may link to it, and the first user may delete the file. The first user is still the owner of the file, but it should be charged to the second user. The simplest reasonably fair algorithm seems to be to spread the charges equally among users who have links to a file. Many installations avoid the issue by not charging any fees at all.

V. PROCESSES AND IMAGES

An *image* is a computer execution environment. It includes a memory image, general register values, status of open files, current directory and the like. An image is the current state of a pseudo-computer.

A *process* is the execution of an image. While the processor is executing on behalf of a process, the image must reside in main memory; during the execution of other processes it remains in main memory unless the appearance of an active, higher-priority process forces it to be swapped out to the disk.

The user-memory part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first hardware protection byte boundary above the program text segment in the virtual address space begins a non-shared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the stack pointer fluctuates.

5.1 Processes

Except while the system is bootstrapping itself into operation, a new process can come into existence only by use of the **fork** system call:

processid = fork ( )

When **fork** is executed, the process splits into two independently executing processes. The two processes have independent copies of the original memory image, and share all open files. The new processes differ only in that one is considered the parent process: in the parent, the returned **processid** actually identifies the child process and is never 0, while in the child, the returned value is always 0.

Because the values returned by **fork** in the parent and child process are distinguishable, each process may determine whether it is the parent or child.

5.2 Pipes

Processes may communicate with related processes using the same system **read** and **write** calls that are used for file-system I/O. The call:

filep = pipe ( )

returns a file descriptor **filep** and creates an inter-process channel called a *pipe*. This channel, like other open files, is passed from parent to child process in the image by the **fork** call. A **read** using a pipe file descriptor waits until another process writes using the file descriptor for the same pipe. At this point, data are passed between the images of the two processes. Neither process need know that a pipe, rather than an ordinary file, is involved.

Although inter-process communication via pipes is a quite valuable tool (see Section 6.2), it is not a completely general mechanism, because the pipe must be set up by a common ancestor of the processes involved.

5.3 Execution of programs

Another major system primitive is invoked by

execute (file, arg_1, arg_2, ... , arg_n)

which requests the system to read in and execute the program named by **file**, passing it string arguments $\mathbf{arg_1}$, $\mathbf{arg_2}$, **...** , $\mathbf{arg_n}$. All the code and data in the process invoking **execute** is replaced from the **file**, but open files, current directory, and inter-process relationships are unaltered. Only if the call fails, for example because **file** could not be found or because its execute-permission bit was not set, does a return take place from the **execute** primitive; it resembles a ''jump'' machine instruction rather than a subroutine call.

5.4 Process synchronization

Another process control system call:

processid = wait (status)

causes its caller to suspend execution until one of its children has completed execution. Then **wait** returns the **processid** of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available.

5.5 Termination

Lastly:

exit (status)

terminates a process, destroys its image, closes its open files, and generally obliterates it. The parent is notified through the **wait** primitive, and **status** is made available to it. Processes may also terminate as a result of various illegal actions or user-generated signals (Section VII below).

VI. THE SHELL

For most users, communication with the system is carried on with the aid of a program called the shell. The shell is a command-line interpreter: it reads lines typed by the user and interprets them as requests to execute other programs. (The shell is described fully elsewhere,[9] so this section will discuss only the theory of its operation.) In simplest form, a command line consists of the command name followed by arguments to the command, all separated by spaces:

command arg_1 arg_2 . . . arg_n

The shell splits up the command name and the arguments into separate strings. Then a file with name **command** is sought; **command** may be a path name including the ‘‘/’’ character to specify any file in the system. If **command** is found, it is brought into memory and executed. The arguments collected by the shell are accessible to the command. When the command is finished, the shell resumes its own execution, and indicates its readiness to accept another command by typing a prompt character.

If file **command** cannot be found, the shell generally prefixes a string such as **/bin/** to **command** and attempts again to find the file. Directory **/bin** contains commands intended to be generally used. (The sequence of directories to be searched may be changed by user request.)

6.1 Standard I/O

The discussion of I/O in Section III above seems to imply that every file used by a program must be opened or created by the program in order to get a file descriptor for the file. Programs executed by the shell, however, start off with three open files with file descriptors 0, 1, and 2. As such a program begins execution, file 1 is open for writing, and is best understood as the standard output file. Except under circumstances indicated below, this file is the user’s terminal. Thus programs that wish to write informative information ordinarily use file descriptor 1. Conversely, file 0 starts off open for reading, and programs that wish to read messages typed by the user read this file.

The shell is able to change the standard assignments of these file descriptors from the user’s terminal printer and keyboard. If one of the arguments to a command is prefixed by ‘‘>’’, file descriptor 1 will, for the duration of the command, refer to the file named after the ‘‘>’’. For example:

ls

ordinarily lists, on the typewriter, the names of the files in the current directory. The command:

ls >there

creates a file called **there** and places the listing there. Thus the argument **>there** means ‘‘place output on **there**.’’ On the other hand:

ed

ordinarily enters the editor, which takes requests from the user via his keyboard. The command

ed <script

interprets **script** as a file of editor commands; thus **<script** means ‘‘take input from **script**.’’

Although the file name following ‘‘<’’ or ‘‘>’’ appears to be an argument to the command, in fact it is interpreted completely by the shell and is not passed to the command at all. Thus no special coding to handle I/O redirection is needed within each command; the command need merely use the standard file descriptors 0 and 1 where appropriate.

File descriptor 2 is, like file 1, ordinarily associated with the terminal output stream. When an output-diversion request with ‘‘>’’ is specified, file 2 remains attached to the terminal, so that commands may produce diagnostic messages that do not silently end up in the output file.

6.2 Filters

An extension of the standard I/O notion is used to direct output from one command to the input of another. A sequence of commands separated by vertical bars causes the shell to execute all the commands simultaneously and to arrange that the standard output of each command be delivered to the standard input of the next command in the sequence. Thus in the command line:

ls | pr −2 | opr

ls lists the names of the files in the current directory; its output is passed to **pr**, which paginates its input with dated headings. (The argument ‘‘−2’’ requests double-column output.) Likewise, the output from **pr** is input to **opr**; this command spools its input onto a file for off-line printing.

This procedure could have been carried out more clumsily by:

```
ls >temp1
pr −2 <temp1 >temp2
opr <temp2
```

followed by removal of the temporary files. In the absence of the ability to redirect output and input, a still clumsier method would have been to require the **ls** command to accept user requests to paginate its output, to print in multi-column format, and to arrange that its output be delivered off-line. Actually it would be surprising, and in fact unwise for efficiency reasons, to expect authors of commands such as **ls** to provide such a wide variety of output options.

A program such as **pr** which copies its standard input to its standard output (with processing) is called a *filter*. Some filters that we have found useful perform character transliteration, selection of lines according to a pattern, sorting of the input, and encryption and decryption.

6.3 Command separators; multitasking

Another feature provided by the shell is relatively straightforward. Commands need not be on different lines; instead they may be separated by semicolons:

```
ls; ed
```

will first list the contents of the current directory, then enter the editor.

A related feature is more interesting. If a command is followed by ‘‘**&**,’’ the shell will not wait for the command to finish before prompting again; instead, it is ready immediately to accept a new command. For example:

```
as source >output &
```

causes **source** to be assembled, with diagnostic output going to **output**; no matter how long the assembly takes, the shell returns immediately. When the shell does not wait for the completion of a command, the identification number of the process running that command is printed. This identification may be used to wait for the completion of the command or to terminate it. The ‘‘**&**’’ may be used several times in a line:

```
as source >output & ls >files &
```

does both the assembly and the listing in the background. In these examples, an output file other than the terminal was provided; if this had not been done, the outputs of the various commands would have been intermingled.

The shell also allows parentheses in the above operations. For example:

```
( date; ls ) >x &
```

writes the current date and time followed by a list of the current directory onto the file **x**. The shell also returns immediately for another request.

6.4 The shell as a command; command files

The shell is itself a command, and may be called recursively. Suppose file **tryout** contains the lines:

```
as source
mv a.out testprog
testprog
```

The **mv** command causes the file **a.out** to be renamed **testprog. a.out** is the (binary) output of the assembler, ready to be executed. Thus if the three lines above were typed on the keyboard, **source** would be assembled, the resulting program renamed **testprog**, and **testprog** executed. When the lines are in **tryout**, the command:

sh <tryout

would cause the shell **sh** to execute the commands sequentially.

The shell has further capabilities, including the ability to substitute parameters and to construct argument lists from a specified subset of the file names in a directory. It also provides general conditional and looping constructions.

6.5 Implementation of the shell

The outline of the operation of the shell can now be understood. Most of the time, the shell is waiting for the user to type a command. When the newline character ending the line is typed, the shell's **read** call returns. The shell analyzes the command line, putting the arguments in a form appropriate for **execute**. Then **fork** is called. The child process, whose code of course is still that of the shell, attempts to perform an **execute** with the appropriate arguments. If successful, this will bring in and start execution of the program whose name was given. Meanwhile, the other process resulting from the **fork**, which is the parent process, **wait**s for the child process to die. When this happens, the shell knows the command is finished, so it types its prompt and reads the keyboard to obtain another command.

Given this framework, the implementation of background processes is trivial; whenever a command line contains ''**&**,'' the shell merely refrains from waiting for the process that it created to execute the command.

Happily, all of this mechanism meshes very nicely with the notion of standard input and output files. When a process is created by the **fork** primitive, it inherits not only the memory image of its parent but also all the files currently open in its parent, including those with file descriptors 0, 1, and 2. The shell, of course, uses these files to read command lines and to write its prompts and diagnostics, and in the ordinary case its children—the command programs—inherit them automatically. When an argument with ''<'' or ''>'' is given, however, the offspring process, just before it performs **execute,** makes the standard I/O file descriptor (0 or 1, respectively) refer to the named file. This is easy because, by agreement, the smallest unused file descriptor is assigned when a new file is **open**ed (or **create**d); it is only necessary to close file 0 (or 1) and open the named file. Because the process in which the command program runs simply terminates when it is through, the association between a file specified after ''<'' or ''>'' and file descriptor 0 or 1 is ended automatically when the process dies. Therefore the shell need not know the actual names of the files that are its own standard input and output, because it need never reopen them.

Filters are straightforward extensions of standard I/O redirection with pipes used instead of files.

In ordinary circumstances, the main loop of the shell never terminates. (The main loop includes the branch of the return from **fork** belonging to the parent process; that is, the branch that does a **wait**, then reads another command line.) The one thing that causes the shell to terminate is discovering an end-of-file condition on its input file. Thus, when the shell is executed as a command with a given input file, as in:

sh <comfile

the commands in **comfile** will be executed until the end of **comfile** is reached; then the instance of the shell invoked by **sh** will terminate. Because this shell process is the child of another instance of the shell, the **wait** executed in the latter will return, and another command may then be processed.

6.6 Initialization

The instances of the shell to which users type commands are themselves children of another process. The last step in the initialization of the system is the creation of a single process and the invocation (via **execute**) of a program called **init**. The role of **init** is to create one process for each terminal channel. The various subinstances of **init** open the appropriate terminals for input and output on files 0, 1, and 2, waiting, if necessary, for carrier to be established on dial-up lines. Then a message is typed out requesting that the user log in. When the user types a name or other identification, the appropriate instance of **init** wakes up, receives the log-in line, and reads a password file. If the user's name is found, and if he is able to supply the correct password, **init** changes to the user's default current directory, sets the process's user ID to that of the person logging in, and performs an **execute** of the shell. At this point, the shell is ready to receive commands and the logging-in protocol is complete.

Meanwhile, the mainstream path of **init** (the parent of all the subinstances of itself that will later become shells) does a **wait**. If one of the child processes terminates, either because a shell found an end of file or because a user typed an incorrect name or password, this path of **init** simply recreates the defunct process, which in turn reopens the appropriate input and output files and types another log-in message. Thus a user may log out simply by typing the end-of-file sequence to the shell.

6.7 Other programs as shell

The shell as described above is designed to allow users full access to the facilities of the system, because it will invoke the execution of any program with appropriate protection mode. Sometimes, however, a different interface to the system is desirable, and this feature is easily arranged for.

Recall that after a user has successfully logged in by supplying a name and password, **init** ordinarily invokes the shell to interpret command lines. The user's entry in the password file may contain the name of a program to be invoked after log-in instead of the shell. This program is free to interpret the user's messages in any way it wishes.

For example, the password file entries for users of a secretarial editing system might specify that the editor **ed** is to be used instead of the shell. Thus when users of the editing system log in, they are inside the editor and can begin work immediately; also, they can be prevented from invoking programs not intended for their use. In practice, it has proved desirable to allow a temporary escape from the editor to execute the formatting program and other utilities.

Several of the games (e.g., chess, blackjack, 3D tic-tac-toe) available on the system illustrate a much more severely restricted environment. For each of these, an entry exists in the password file specifying that the appropriate game-playing program is to be invoked instead of the shell. People who log in as a player of one of these games find themselves limited to the game and unable to investigate the (presumably more interesting) offerings of the UNIX system as a whole.

VII. TRAPS

The PDP-11 hardware detects a number of program faults, such as references to non-existent memory, unimplemented instructions, and odd addresses used where an even address is required. Such faults cause the processor to trap to a system routine. Unless other arrangements have been made, an illegal action causes the system to terminate the process and to write its image on file **core** in the current directory. A debugger can be used to determine the state of the program at the time of the fault.

Programs that are looping, that produce unwanted output, or about which the user has second thoughts may be halted by the use of the **interrupt** signal, which is generated by typing the ''delete'' character. Unless special action has been taken, this signal simply causes the program to cease execution without producing a **core** file. There is also a **quit** signal used to force an image file to be produced. Thus programs that loop unexpectedly may be halted and the remains inspected without prearrangement.

The hardware-generated faults and the interrupt and quit signals can, by request, be either ignored or caught by a process. For example, the shell ignores quits to prevent a quit from logging the user out. The editor catches interrupts and returns to its command level. This is useful for stopping long printouts without losing work in progress (the editor manipulates a copy of the file it is editing). In systems without floating-point hardware, unimplemented instructions are caught and floating-point instructions are interpreted.

VIII. PERSPECTIVE

Perhaps paradoxically, the success of the UNIX system is largely due to the fact that it was not designed to meet any predefined objectives. The first version was written when one of us (Thompson), dissatisfied with the available computer facilities, discovered a little-used PDP-7 and set out to create a more hospitable environment. This (essentially personal) effort was sufficiently successful to gain the interest of the other author and several colleagues, and later to justify the acquisition of the PDP-11/20, specifically to support a text editing and formatting system. When in turn the 11/20 was outgrown, the system had proved useful enough to persuade management to invest in the PDP-11/45, and later in the PDP-11/70 and Interdata 8/32 machines, upon which it developed to its present form. Our goals throughout the effort, when articulated at all, have always been to build a comfortable relationship with the machine and to explore ideas and inventions in operating systems and other software. We have not been faced with the need to satisfy someone else's requirements, and for this freedom we are grateful.

Three considerations that influenced the design of UNIX are visible in retrospect.

First: because we are programmers, we naturally designed the system to make it easy to write, test, and run programs. The most important expression of our desire for programming convenience was that the system was arranged for interactive use, even though the original version only supported one user. We believe that a properly designed interactive system is much more productive and satisfying to use than a ''batch'' system. Moreover, such a system is rather easily adaptable to noninteractive use, while the converse is not true.

Second: there have always been fairly severe size constraints on the system and its software. Given the partially antagonistic desires for reasonable efficiency and expressive power, the size constraint has encouraged not only economy, but also a certain elegance of design. This may be a thinly disguised version of the ''salvation through suffering'' philosophy, but in our case it worked.

Third: nearly from the start, the system was able to, and did, maintain itself. This fact is more important than it might seem. If designers of a system are forced to use that system, they quickly become aware of its functional and superficial deficiencies and are strongly motivated to correct them before it is too late. Because all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others.

The aspects of UNIX discussed in this paper exhibit clearly at least the first two of these design considerations. The interface to the file system, for example, is extremely convenient from a programming standpoint. The lowest possible interface level is designed to eliminate distinctions between the various devices and files and between direct and sequential access. No large ''access method'' routines are required to insulate the programmer from the system calls; in fact, all user programs either call the system directly or use a small library program, less than a page long, that buffers a number of characters and reads or writes them all at once.

Another important aspect of programming convenience is that there are no ''control blocks'' with a complicated structure partially maintained by and depended on by the file system or other system calls. Generally speaking, the contents of a program's address space are the property of the program, and we have tried to avoid placing restrictions on the data structures within that address space.

Given the requirement that all programs should be usable with any file or device as input or output, it is also desirable to push device-dependent considerations into the operating system itself. The only alternatives seem to be to load, with all programs, routines for dealing with each device, which is expensive in space, or to depend on some means of dynamically linking to the routine appropriate to each device when it is actually needed, which is expensive either in overhead or in hardware.

Likewise, the process-control scheme and the command interface have proved both convenient and efficient. Because the shell operates as an ordinary, swappable user program, it consumes no ''wired-down'' space in the system proper, and it may be made as powerful as desired at little cost. In particular, given the framework in which the shell executes as a process that spawns other processes to perform commands, the notions of I/O redirection, background processes, command files, and user-selectable system interfaces all become essentially trivial to implement.

Influences

The success of UNIX lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas, and especially in showing that they can be keys to the implementation of a small yet powerful operating system.

The **fork** operation, essentially as we implemented it, was present in the GENIE time-sharing system.[10] On a number of points we were influenced by Multics, which suggested the particular form of the I/O system calls[11] and both the name of the shell and its general functions. The notion that the shell should create a process for each command was also suggested to us by the early design of Multics, although in that system it was later dropped for efficiency reasons. A similar scheme is used by TENEX.[12]

IX. STATISTICS

The following numbers are presented to suggest the scale of the Research UNIX operation. Those of our users not involved in document preparation tend to use the system for program development, especially language work. There are few important ''applications'' programs.

Overall, we have today:

125	user population
33	maximum simultaneous users
1,630	directories
28,300	files
301,700	512-byte secondary storage blocks used

There is a ''background'' process that runs at the lowest possible priority; it is used to soak up any idle CPU time. It has been used to produce a million-digit approximation to the constant *e*, and other semi-infinite problems. Not counting this background work, we average daily:

13,500	commands
9.6	CPU hours
230	connect hours
62	different users
240	log-ins

X. ACKNOWLEDGMENTS

The contributors to UNIX are, in the traditional but here especially apposite phrase, too numerous to mention. Certainly, collective salutes are due to our colleagues in the Computing Science Research Center. R. H. Canaday contributed much to the basic design of the file system. We are particularly appreciative of the inventiveness, thoughtful criticism, and constant support of R. Morris, M. D. McIlroy, and J. F. Ossanna.

References

1. L. P. Deutsch and B. W. Lampson, ''An online editor,'' *Comm. Assoc. Comp. Mach.* **10**(12), pp.793-799, 803 (December 1967).
2. B. W. Kernighan and L. L. Cherry, ''A System for Typesetting Mathematics,'' *Comm. Assoc. Comp. Mach.* **18**, pp.151-157 (March 1975).
3. B. W. Kernighan, M. E. Lesk, and J. F. Ossanna, ''UNIX Time-Sharing System: Document Preparation,'' *Bell Sys. Tech. J.* **57**(6), pp.2115-2135 (1978).

4. T. A. Dolotta and J. R. Mashey, ''An Introduction to the Programmer's Workbench,'' *Proc. 2nd Int. Conf. on Software Engineering*, pp.164-168 (October 13-15, 1976).

5. T. A. Dolotta, R. C. Haight, and J. R. Mashey, ''UNIX Time-Sharing System: The Programmer's Workbench,'' *Bell Sys. Tech. J.* **57**(6), pp.2177-2200 (1978).

6. H. Lycklama, ''UNIX Time-Sharing System: UNIX on a Microprocessor,'' *Bell Sys. Tech. J.* **57**(6), pp.2087-2101 (1978).

7. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).

8. Aleph-null, ''Computer Recreations,'' *Software Practice and Experience* **1**(2), pp.201-204 (April-June 1971).

9. S. R. Bourne, ''UNIX Time-Sharing System: The UNIX Shell,'' *Bell Sys. Tech. J.* **57**(6), pp.1971-1990 (1978).

10. L. P. Deutsch and B. W. Lampson, ''SDS 930 time-sharing system preliminary reference manual,'' Doc. 30.10.10, Project GENIE, Univ. Cal. at Berkeley (April 1965).

11. R. J. Feiertag and E. I. Organick, ''The Multics input-output system,'' *Proc. Third Symposium on Operating Systems Principles*, pp.35-41 (October 18-20, 1971).

12. D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, ''TENEX, a Paged Time Sharing System for the PDP-10,'' *Comm. Assoc. Comp. Mach.* **15**(3), pp.135-143 (March 1972).

UNIX For Beginners — Second Edition

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is meant to help new users get started on the UNIX† operating system. It includes:

- basics needed for day-to-day use of the system — typing commands, correcting typing mistakes, logging in and out, mail, inter-terminal communication, the file system, printing files, redirecting I/O, pipes, and the shell.
- document preparation — a brief discussion of the major formatting programs and macro packages, hints on preparing documents, and capsule descriptions of some supporting software.
- UNIX programming — using the editor, programming the shell, programming in C, other languages and tools.
- An annotated UNIX bibliography.

October 2, 1978

†UNIX is a Trademark of Bell Laboratories.

UNIX For Beginners — Second Edition

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

INTRODUCTION

From the user's point of view, the UNIX operating system is easy to learn and use, and presents few of the usual impediments to getting the job done. It is hard, however, for the beginner to know where to start, and how to make the best use of the facilities available. The purpose of this introduction is to help new users get used to the main ideas of the UNIX system and start making effective use of it quickly.

You should have a couple of other documents with you for easy reference as you read this one. The most important is *The UNIX Programmer's Manual*; it's often easier to tell you to read about something in the manual than to repeat its contents here. The other useful document is *A Tutorial Introduction to the UNIX Text Editor,* which will tell you how to use the editor to get text — programs, data, documents — into the computer.

A word of warning: the UNIX system has become quite popular, and there are several major variants in widespread use. Of course details also change with time. So although the basic structure of UNIX and how to use it is common to all versions, there will certainly be a few things which are different on your system from what is described here. We have tried to minimize the problem, but be aware of it. In cases of doubt, this paper describes Version 7 UNIX.

This paper has five sections:

1. Getting Started: How to log in, how to type, what to do about mistakes in typing, how to log out. Some of this is dependent on which system you log into (phone numbers, for example) and what terminal you use, so this section must necessarily be supplemented by local information.
2. Day-to-day Use: Things you need every day to use the system effectively: generally useful commands; the file system.
3. Document Preparation: Preparing manuscripts is one of the most common uses for UNIX systems. This section contains advice, but not extensive instructions on any of the formatting tools.
4. Writing Programs: UNIX is an excellent system for developing programs. This section talks about some of the tools, but again is not a tutorial in any of the programming languages provided by the system.
5. A UNIX Reading List. An annotated bibliography of documents that new users should be aware of.

I. GETTING STARTED

Logging In

You must have a UNIX login name, which you can get from whoever administers your system. You also need to know the phone number, unless your system uses permanently connected terminals. The UNIX system is capable of dealing with a wide variety of terminals: Terminet 300's; Execuport, TI and similar portables; video (CRT) terminals like the HP2640, etc.; high-priced graphics terminals like the Tektronix 4014; plotting terminals like those from GSI and DASI; and even the venerable Teletype in its various forms. But note: UNIX is strongly oriented towards devices with *lower case.* If your terminal produces only upper case (e.g., model 33 Teletype, some video and portable terminals), life will be so difficult that you should look for another terminal.

Be sure to set the switches appropriately on your device. Switches that might need to be adjusted include the speed, upper/lower case mode, full duplex, even parity, and any others that local wisdom advises. Establish a connection using whatever magic is needed for your terminal; this may involve dialing a telephone call or merely flipping a switch. In either case, UNIX should type ‘‘**login:**’’ at you. If it types garbage, you may be at the wrong speed; check the switches. If that fails, push the ‘‘break’’ or ‘‘interrupt’’ key a few times, slowly. If that fails to produce a login message, consult a guru.

When you get a **login:** message, type your login name *in lower case.* Follow it by a RETURN; the system will not do anything until you type a RETURN. If a password is required, you will be asked for it, and (if possible) printing will be turned off while you type it. Don't forget RETURN.

The culmination of your login efforts is a ''prompt character,'' a single character that indicates that the system is ready to accept commands from you. The prompt character is usually a dollar sign **$** or a percent sign **%**. (You may also get a message of the day just before the prompt character, or a notification that you have mail.)

Typing Commands

Once you've seen the prompt character, you can type commands, which are requests that the system do something. Try typing

date

followed by RETURN. You should get back something like

Mon Jan 16 14:17:10 EST 1978

Don't forget the RETURN after the command, or nothing will happen. If you think you're being ignored, type a RETURN; something should happen. RETURN won't be mentioned again, but don't forget it — it has to be there at the end of each line.

Another command you might try is **who**, which tells you everyone who is currently logged in:

who

gives something like

mb	**tty01**	**Jan 16**	**09:11**
ski	**tty05**	**Jan 16**	**09:33**
gam	**tty11**	**Jan 16**	**13:07**

The time is when the user logged in; ''ttyxx'' is the system's idea of what terminal the user is on.

If you make a mistake typing the command name, and refer to a non-existent command, you will be told. For example, if you type

whom

you will be told

whom: not found

Of course, if you inadvertently type the name of some other command, it will run, with more or less mysterious results.

Strange Terminal Behavior

Sometimes you can get into a state where your terminal acts strangely. For example, each letter may be typed twice, or the RETURN may not cause a line feed or a return to the left margin. You can often fix this by logging out and logging back in. Or you can read the description of the command **stty** in section I of the manual. To get intelligent treatment of tab characters (which are much used in UNIX) if your terminal doesn't have tabs, type the command

stty −tabs

and the system will convert each tab into the right number of blanks for you. If your terminal does have computer-settable tabs, the command **tabs** will set the stops correctly for you.

Mistakes in Typing

If you make a typing mistake, and see it before RETURN has been typed, there are two ways to recover. The sharp-character # erases the last character typed; in fact successive uses of # erase characters back to the beginning of the line (but not beyond). So if you type badly, you can correct as you go:

dd#atte##e

is the same as **date**.

The at-sign @ erases all of the characters typed so far on the current input line, so if the line is irretrievably fouled up, type an @ and start the line over.

What if you must enter a sharp or at-sign as part of the text? If you precede either # or @ by a backslash \, it loses its erase meaning. So to enter a sharp or at-sign in something, type \# or \@. The system will always echo a newline at you after your at-sign, even if preceded by a backslash. Don't worry — the at-sign has been recorded.

To erase a backslash, you have to type two sharps or two at-signs, as in \##. The backslash is used extensively in UNIX to indicate that the following character is in some way special.

Read-ahead

UNIX has full read-ahead, which means that you can type as fast as you want, whenever you want, even when some command is typing at you. If you type during output, your input characters will appear intermixed with the output characters, but they will be stored away and interpreted in the correct order. So you can type several commands one after another without waiting for the first to finish or even begin.

Stopping a Program

You can stop most programs by typing the character ''DEL'' (perhaps called ''delete'' or ''rubout'' on your terminal). The ''interrupt'' or ''break'' key found on most terminals can also be used. In a few programs, like the text editor, DEL stops whatever the program is doing but leaves you in that program. Hanging up the phone will stop most programs.

Logging Out

The easiest way to log out is to hang up the phone. You can also type

login

and let someone else use the terminal you were on. It is usually not sufficient just to turn off the terminal. Most UNIX systems do not use a time-out mechanism, so you'll be there forever unless you hang up.

Mail

When you log in, you may sometimes get the message

You have mail.

UNIX provides a postal system so you can communicate with other users of the system. To read your mail, type the command

mail

Your mail will be printed, one message at a time, most recent message first. After each message, **mail** waits for you to say what to do with it. The two basic responses are **d**, which deletes the message, and RETURN, which does not (so it will still be there the next time you read your mailbox). Other responses are described in the manual. (Earlier versions of **mail** do not process one message at a time, but are otherwise similar.)

How do you send mail to someone else? Suppose it is to go to ''joe'' (assuming ''joe'' is someone's login name). The easiest way is this:

mail joe
now type in the text of the letter
on as many lines as you like ...
After the last line of the letter
type the character ''control–d'',
that is, hold down ''control'' and type
a letter ''d''.

And that's it. The ''control-d'' sequence, often called ''EOF'' for end-of-file, is used throughout the system to mark the end of input from a terminal, so you might as well get used to it.

For practice, send mail to yourself. (This isn't as strange as it might sound — mail to oneself is a handy reminder mechanism.)

There are other ways to send mail — you can send a previously prepared letter, and you can mail to a number of people all at once. For more details see **mail**(1). (The notation **mail**(1) means the command **mail** in section 1 of the *UNIX Programmer's Manual*.)

Writing to other users

At some point, out of the blue will come a message like

Message from joe tty07...

accompanied by a startling beep. It means that Joe wants to talk to you, but unless you take explicit action you won't be able to talk back. To respond, type the command

write joe

This establishes a two-way communication path. Now whatever Joe types on his terminal will appear on yours and vice versa. The path is slow, rather like talking to the moon. (If you are in the middle of something, you have to get to a state where you can type a command. Normally, whatever program you are running has to terminate or be terminated. If you're editing, you can escape temporarily from the editor — read the editor tutorial.)

A protocol is needed to keep what you type from getting garbled up with what Joe types. Typically it's like this:

Joe types **write smith** and waits.
Smith types **write joe** and waits.
Joe now types his message (as many lines as he likes). When he's ready for a reply, he signals it by typing **(o)**, which stands for ''over''.
Now Smith types a reply, also terminated by **(o)**.
This cycle repeats until someone gets tired; he then signals his intent to quit with **(oo)**, for ''over and out''.
To terminate the conversation, each side must type a ''control-d'' character alone on a line. (''Delete'' also works.) When the other person types his ''control-d'', you will get the message **EOF** on your terminal.

If you write to someone who isn't logged in, or who doesn't want to be disturbed, you'll be told. If the target is logged in but doesn't answer after a decent interval, simply type ''control-d''.

On-line Manual

The *UNIX Programmer's Manual* is typically kept on-line. If you get stuck on something, and can't find an expert to assist you, you can print on your terminal some manual section that might help. This is also useful for getting the most up-to-date information on a command. To print a manual section, type ''man command-name''. Thus to read up on the **who** command, type

man who

and, of course,

man man

tells all about the **man** command.

Computer Aided Instruction

Your UNIX system may have available a program called **learn**, which provides computer aided instruction on the file system and basic commands, the editor, document preparation, and even C programming. Try typing the command

learn

If **learn** exists on your system, it will tell you what to do from there.

II. DAY-TO-DAY USE

Creating Files — The Editor

If you have to type a paper or a letter or a program, how do you get the information stored in the machine? Most of these tasks are done with the UNIX ''text editor'' **ed**. Since **ed** is thoroughly documented in **ed**(1) and explained in *A Tutorial Introduction to the UNIX Text Editor,* we won't spend any time here describing how to use it. All we want it for right now is to make some *files.* (A file is just a collection of information stored in the machine, a simplistic but adequate definition.)

To create a file called **junk** with some text in it, do the following:

ed junk	(invokes the text editor)
a	(command to ''ed'', to add text)
now type in	
whatever text you want ...	
.	(signals the end of adding text)

The ''**.**'' that signals the end of adding text must be at the beginning of a line by itself. Don't forget it, for until it is typed, no other **ed** commands will be recognized — everything you type will be treated as text to be added.

At this point you can do various editing operations on the text you typed in, such as correcting spelling mistakes, rearranging paragraphs and the like. Finally, you must write the information you have typed into a file with the editor command **w**:

w

ed will respond with the number of characters it wrote into the file **junk**.

Until the **w** command, nothing is stored permanently, so if you hang up and go home the information is lost.† But after **w** the information is there permanently; you can re-access it any time by typing

ed junk

Type a **q** command to quit the editor. (If you try to quit without writing, **ed** will print a **?** to remind you. A second **q** gets you out regardless.)

Now create a second file called **temp** in the same manner. You should now have two files, **junk** and **temp**.

What files are out there?

The **ls** (for ''list'') command lists the names (not contents) of any of the files that UNIX knows about. If you type

ls

the response will be

junk
temp

which are indeed the two files just created. The names are sorted into alphabetical order automatically, but other variations are possible. For example, the command

ls –t

causes the files to be listed in the order in which they were last changed, most recent first. The **–l** option gives a ''long'' listing:

ls –l

will produce something like

```
–rw–rw–rw–  1 bwk   41 Jul 22 2:56 junk
–rw–rw–rw–  1 bwk   78 Jul 22 2:57 temp
```

The date and time are of the last change to the file. The 41 and 78 are the number of characters (which should agree with the numbers you got from **ed**). **bwk** is the owner of the file, that is, the person who created it. The **–rw–rw–rw–** tells who has permission to read and write the file, in this case everyone.

Options can be combined: **ls –lt** gives the same thing as **ls –l**, but sorted into time order. You can also name the files you're interested in, and **ls** will list the information about them only. More details can be found in **ls**(1).

The use of optional arguments that begin with a minus sign, like **–t** and **–lt**, is a common convention for UNIX programs. In general, if a program accepts such optional arguments, they precede any filename arguments. It is also vital that you separate the various arguments with spaces: **ls–l** is not the same as **ls –l**.

Printing Files

Now that you've got a file of text, how do you print it so people can look at it? There are a host of programs that do that, probably more than are needed.

One simple thing is to use the editor, since printing is often done just before making changes anyway. You can say

ed junk
1,$p

ed will reply with the count of the characters in **junk** and then print all the lines in the file. After you learn how to use the editor, you can be selective about the parts you print.

There are times when it's not feasible to use the editor for printing. For example, there is a limit on how big a file **ed** can handle (several thousand lines). Secondly, it will only print one file at a time, and

† This is not strictly true — if you hang up while editing, the data you were working on is saved in a file called **ed.hup**, which you can continue with at your next session.

sometimes you want to print several, one after another. So here are a couple of alternatives.

First is **cat**, the simplest of all the printing programs. **cat** simply prints on the terminal the contents of all the files named in a list. Thus

cat junk

prints one file, and

cat junk temp

prints two. The files are simply concatenated (hence the name ''**cat**'') onto the terminal.

pr produces formatted printouts of files. As with **cat**, **pr** prints all the files named in a list. The difference is that it produces headings with date, time, page number and file name at the top of each page, and extra lines to skip over the fold in the paper. Thus,

pr junk temp

will print **junk** neatly, then skip to the top of a new page and print **temp** neatly.

pr can also produce multi-column output:

pr −3 junk

prints **junk** in 3-column format. You can use any reasonable number in place of ''3'' and **pr** will do its best. **pr** has other capabilities as well; see **pr**(1).

It should be noted that **pr** is *not* a formatting program in the sense of shuffling lines around and justifying margins. The true formatters are **nroff** and **troff**, which we will get to in the section on document preparation.

There are also programs that print files on a high-speed printer. Look in your manual under **opr** and **lpr**. Which to use depends on what equipment is attached to your machine.

Shuffling Files About

Now that you have some files in the file system and some experience in printing them, you can try bigger things. For example, you can move a file from one place to another (which amounts to giving it a new name), like this:

mv junk precious

This means that what used to be ''junk'' is now ''precious''. If you do an **ls** command now, you will get

precious
temp

Beware that if you move a file to another one that already exists, the already existing contents are lost forever.

If you want to make a *copy* of a file (that is, to have two versions of something), you can use the **cp** command:

cp precious temp1

makes a duplicate copy of **precious** in **temp1**.

Finally, when you get tired of creating and moving files, there is a command to remove files from the file system, called **rm**.

rm temp temp1

will remove both of the files named.

You will get a warning message if one of the named files wasn't there, but otherwise **rm**, like most UNIX commands, does its work silently. There is no prompting or chatter, and error messages are occasionally curt. This terseness is sometimes disconcerting to newcomers, but experienced users find it desirable.

What's in a Filename

So far we have used filenames without ever saying what's a legal name, so it's time for a couple of rules. First, filenames are limited to 14 characters, which is enough to be descriptive. Second, although you can use almost any character in a filename, common sense says you should stick to ones that are visible, and that you should probably avoid characters that might be used with other meanings. We have already seen, for example, that in the **ls** command, **ls −t** means to list in time order. So if you had a file whose name was **−t**, you would have a tough time listing it by name. Besides the minus sign, there are other characters which have special meaning. To avoid pitfalls, you would do well to use only letters, numbers and the period until you're familiar with the situation.

On to some more positive suggestions. Suppose you're typing a large document like a book. Logically this divides into many small pieces, like chapters and perhaps sections. Physically it must be divided too, for **ed** will not handle really big files. Thus you should type the document as a number of files. You might have a separate file for each chapter, called

chap1
chap2
etc...

Or, if each chapter were broken into several files, you might have

chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...

You can now tell at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice UNIX user. What if you wanted to print the whole book? You could say

pr chap1.1 chap1.2 chap1.3 ......

but you would get tired pretty fast, and would probably even make mistakes. Fortunately, there is a shortcut. You can say

pr chap*

The * means ''anything at all,'' so this translates into ''print all files whose names begin with **chap**'', listed in alphabetical order.

This shorthand notation is not a property of the **pr** command, by the way. It is system-wide, a service of the program that interprets commands (the ''shell,'' **sh**(1)). Using that fact, you can see how to list the names of the files in the book:

ls chap*

produces

chap1.1
chap1.2
chap1.3
...

The * is not limited to the last position in a filename — it can be anywhere and can occur several times. Thus

rm *junk* *temp*

removes all files that contain **junk** or **temp** as any part of their name. As a special case, * by itself matches every filename, so

pr *

prints all your files (alphabetical order), and

rm *

removes *all files.* (You had better be *very* sure that's what you wanted to say!)

The * is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4 and 9. Then you can say

pr chap[12349]*

The **[...]** means to match any of the characters inside the brackets. A range of consecutive letters or digits can be abbreviated, so you can also do this with

pr chap[1−49]*

Letters can also be used within brackets: **[a−z]** matches any character in the range **a** through **z**.

The **?** pattern matches any single character, so

ls ?

lists all files which have single-character names, and

ls −l chap?.1

lists information about the first file of each chapter (**chap1.1**, **chap2.1**, etc.).

Of these niceties, * is certainly the most useful, and you should get used to it. The others are frills, but worth knowing.

If you should ever have to turn off the special meaning of *, **?**, etc., enclose the entire argument in single quotes, as in

ls ′?′

We'll see some more examples of this shortly.

What's in a Filename, Continued

When you first made that file called **junk**, how did the system know that there wasn't another **junk** somewhere else, especially since the person in the next office is also reading this tutorial? The answer is that generally each user has a private *directory*, which contains only the files that belong to him. When you log in, you are ''in'' your directory. Unless you take special action, when you create a new file, it is made in the directory that you are currently in; this is most often your own directory, and thus the file is unrelated to any other file of the same name that might exist in someone else's directory.

The set of all files is organized into a (usually big) tree, with your files located several branches into the tree. It is possible for you to ''walk'' around this tree, and to find any file in the system, by starting at the root of the tree and walking along the proper set of branches. Conversely, you can start where you are and walk toward the root.

Let's try the latter first. The basic tools is the command **pwd** (''print working directory''), which prints the name of the directory you are currently in.

Although the details will vary according to the system you are on, if you give the command **pwd**, it will print something like

/usr/your-name

This says that you are currently in the directory **your-name**, which is in turn in the directory **/usr**, which is in turn in the root directory called by convention just **/**. (Even if it's not called **/usr** on your system, you will get something analogous. Make the corresponding changes and read on.)

If you now type

ls /usr/your-name

you should get exactly the same list of file names as you get from a plain **ls**: with no arguments, **ls** lists the contents of the current directory; given the name of a directory, it lists the contents of that directory.

Next, try

ls /usr

This should print a long series of names, among which is your own login name **your-name**. On many systems, **usr** is a directory that contains the directories of all the normal users of the system, like you.

The next step is to try

ls /

You should get a response something like this (although again the details may be different):

bin
dev
etc
lib
tmp
usr

This is a collection of the basic directories of files that the system knows about; we are at the root of the tree.

Now try

cat /usr/your-name/junk

(if **junk** is still around in your directory). The name

/usr/your-name/junk

is called the **pathname** of the file that you normally think of as ''junk''. ''Pathname'' has an obvious meaning: it represents the full name of the path you have to follow from the root through the tree of directories to get to a particular file. It is a universal rule in the UNIX system that anywhere you can use an ordinary filename, you can use a pathname.

Here is a picture which may make this clearer:

```
                 (root)
                 / | \
                /  |  \
               /   |   \
     bin     etc    usr    dev    tmp
    / | \   / | \  / | \  / | \  / | \
                  /  |  \
                 /   |   \
          adam     eve    mary
          /       /  \       \
                 /    \      junk
              junk  temp
```

Notice that Mary's **junk** is unrelated to Eve's.

This isn't too exciting if all the files of interest are in your own directory, but if you work with someone else or on several projects concurrently, it becomes handy indeed. For example, your friends can print your book by saying

pr /usr/your-name/chap*

Similarly, you can find out what files your neighbor has by saying

ls /usr/neighbor-name

or make your own copy of one of his files by

cp /usr/your-neighbor/his-file yourfile

If your neighbor doesn't want you poking around in his files, or vice versa, privacy can be arranged. Each file and directory has read-write-execute permissions for the owner, a group, and everyone else, which can be set to control access. See **ls**(1) and **chmod**(1) for details. As a matter of observed fact, most users most of the time find openness of more benefit than privacy.

As a final experiment with pathnames, try

ls /bin /usr/bin

Do some of the names look familiar? When you run a program, by typing its name after the prompt character, the system simply looks for a file of that name. It normally looks first in your directory (where it typically doesn't find it), then in **/bin** and finally in **/usr/bin**. There is nothing magic about commands like **cat** or **ls**, except that they have been collected into a couple of places to be easy to find and administer.

What if you work regularly with someone else on common information in his directory? You could just log in as your friend each time you want to, but you can also say ''I want to work on his files instead of my own''. This is done by changing the directory that you are currently in:

cd /usr/your-friend

(On some systems, **cd** is spelled **chdir**.) Now when you use a filename in something like **cat** or **pr**, it refers to the file in your friend's directory. Changing directories doesn't affect any permissions associated with a file — if you couldn't access a file from your own directory, changing to another directory won't alter that fact. Of course, if you forget what directory you're in, type

pwd

to find out.

It is usually convenient to arrange your own files so that all the files related to one thing are in a directory separate from other projects. For example, when you write your book, you might want to keep all the text in a directory called **book**. So make one with

mkdir book

then go to it with

cd book

then start typing chapters. The book is now found in (presumably)

/usr/your-name/book

To remove the directory **book**, type

rm book/*
rmdir book

The first command removes all files from the directory; the second removes the empty directory.

You can go up one level in the tree of files by saying

cd ..

‘‘**..**’’ is the name of the parent of whatever directory you are currently in. For completeness, ‘‘**.**’’ is an alternate name for the directory you are in.

Using Files instead of the Terminal

Most of the commands we have seen so far produce output on the terminal; some, like the editor, also take their input from the terminal. It is universal in UNIX systems that the terminal can be replaced by a file for either or both of input and output. As one example,

ls

makes a list of files on your terminal. But if you say

ls >filelist

a list of your files will be placed in the file **filelist** (which will be created if it doesn't already exist, or overwritten if it does). The symbol > means ‘‘put the output on the following file, rather than on the terminal.’’ Nothing is produced on the terminal. As another example, you could combine several files into one by capturing the output of **cat** in a file:

cat f1 f2 f3 >temp

The symbol >> operates very much like > does, except that it means ‘‘add to the end of.’’ That is,

cat f1 f2 f3 >>temp

means to concatenate **f1**, **f2** and **f3** to the end of whatever is already in **temp**, instead of overwriting the existing contents. As with >, if **temp** doesn't exist, it will be created for you.

In a similar way, the symbol < means to take the input for a program from the following file, instead of from the terminal. Thus, you could make up a script of commonly used editing commands and put them into a file called **script**. Then you can run the script on a file by saying

ed file <script

As another example, you can use **ed** to prepare a letter in file **let**, then send it to several people with

mail adam eve mary joe <let

Pipes

One of the novel contributions of the UNIX system is the idea of a *pipe*. A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes — a pipeline.

For example,

pr f g h

will print the files **f**, **g**, and **h**, beginning each on a new page. Suppose you want them run together instead. You could say

cat f g h >temp
pr <temp
rm temp

but this is more work than necessary. Clearly what we want is to take the output of **cat** and connect it to the input of **pr**. So let us use a pipe:

cat f g h | pr

The vertical bar | means to take the output from **cat**, which would normally have gone to the terminal, and put it into **pr** to be neatly formatted.

There are many other examples of pipes. For example,

ls | pr −3

prints a list of your files in three columns. The program **wc** counts the number of lines, words and characters in its input, and as we saw earlier, **who** prints a list of currently-logged on people, one per line. Thus

who | wc

tells how many people are logged on. And of course

ls | wc

counts your files.

Any program that reads from the terminal can read from a pipe instead; any program that writes on the terminal can drive a pipe. You can have as many elements in a pipeline as you wish.

Many UNIX programs are written so that they will take their input from one or more files if file arguments are given; if no arguments are given they will read from the terminal, and thus can be used in pipelines. **pr** is one example:

pr −3 a b c

prints files **a**, **b** and **c** in order in three columns. But in

cat a b c | pr −3

pr prints the information coming down the pipeline, still in three columns.

The Shell

We have already mentioned once or twice the mysterious ‘‘shell,’’ which is in fact **sh**(1). The shell is the program that interprets what you type as commands and arguments. It also looks after translating *, etc., into lists of filenames, and <, >, and | into changes of input and output streams.

The shell has other capabilities too. For example, you can run two programs with one command line by separating the commands with a semicolon; the shell recognizes the semicolon and breaks the line into two commands. Thus

date; who

does both commands before returning with a prompt character.

You can also have more than one program running *simultaneously* if you wish. For example, if you are doing something time-consuming, like the editor script of an earlier section, and you don’t want to wait around for the results before starting something else, you can say

ed file <script &

The ampersand at the end of a command line says ‘‘start this command running, then take further commands from the terminal immediately,’’ that is, don’t wait for it to complete. Thus the script will begin, but you can do something else at the same time. Of course, to keep the output from interfering with what you’re doing on the terminal, it would be better to say

ed file <script >script.out &

which saves the output lines in a file called **script.out**.

When you initiate a command with **&**, the system replies with a number called the process number, which identifies the command in case you later want to stop it. If you do, you can say

kill process-number

If you forget the process number, the command **ps** will tell you about everything you have running. (If you are desperate, **kill 0** will kill all your processes.) And if you’re curious about other people, **ps a** will tell you about *all* programs that are currently running.

You can say

(command-1; command-2; command-3) &

to start three commands in the background, or you can start a background pipeline with

command-1 | command-2 &

Just as you can tell the editor or some similar program to take its input from a file instead of from the terminal, you can tell the shell to read a file to get commands. (Why not? The shell, after all, is just a program, albeit a clever one.) For instance, suppose you want to set tabs on your terminal, and find out the date and who’s on the system every time you log in. Then you can put the three necessary commands (**tabs**, **date**, **who**) into a file, let’s call it **startup**, and then run it with

sh startup

This says to run the shell with the file **startup** as input. The effect is as if you had typed the contents of **startup** on the terminal.

If this is to be a regular thing, you can eliminate the need to type **sh**: simply type, once only, the command

chmod +x startup

and thereafter you need only say

startup

to run the sequence of commands. The **chmod**(1) command marks the file executable; the shell recognizes this and runs it as a sequence of commands.

If you want **startup** to run automatically every time you log in, create a file in your login directory called **.profile**, and place in it the line **startup**. When the shell first gains control when you log in, it looks for the **.profile** file and does whatever commands it finds in it. We’ll get back to the shell in the section on programming.

III. DOCUMENT PREPARATION

UNIX systems are used extensively for document preparation. There are two major formatting programs, that is, programs that produce a text with justified right margins, automatic page numbering and titling, automatic hyphenation, and the like. **nroff** is designed to produce output on terminals and line-printers. **troff** (pronounced ‘‘tee-roff’’) instead drives a phototypesetter, which produces very high quality output on photographic paper. This paper was formatted with **troff**.

Formatting Packages

The basic idea of **nroff** and **troff** is that the text to be formatted contains within it ‘‘formatting commands’’ that indicate in detail how the formatted text is to look. For example, there might be commands that specify how long lines are, whether to use single or double spacing, and what running titles to use on each page.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several ‘‘packages’’ of canned formatting requests are available to let you specify paragraphs, running titles, footnotes, multi-column output, and so on, with little effort and without having to learn **nroff** and **troff**. These packages take a modest effort to learn, but the rewards for using them are so great that it is time well spent.

In this section, we will provide a hasty look at the ‘‘manuscript’’ package known as **−ms**. Formatting requests typically consist of a period and two upper-case letters, such as **.TL**, which is used to introduce a title, or **.PP** to begin a new paragraph.

A document is typed so it looks something like this:

```
.TL
title of document
.AU
author name
.SH
section heading
.PP
paragraph ...
.PP
another paragraph ...
.SH
another section heading
.PP
etc.
```

The lines that begin with a period are the formatting requests. For example, **.PP** calls for starting a new paragraph. The precise meaning of **.PP** depends on what output device is being used (typesetter or terminal, for instance), and on what publication the document will appear in. For example, **−ms** normally assumes that a paragraph is preceded by a space (one line in **nroff**, ½ line in **troff**), and the first word is indented. These rules can be changed if you like, but they are changed by changing the interpretation of **.PP**, not by re-typing the document.

To actually produce a document in standard format using **−ms**, use the command

troff −ms files ...

for the typesetter, and

nroff −ms files ...

for a terminal. The **−ms** argument tells **troff** and **nroff** to use the manuscript package of formatting requests.

There are several similar packages; check with a local expert to determine which ones are in common use on your machine.

Supporting Tools

In addition to the basic formatters, there is a host of supporting programs that help with document preparation. The list in the next few paragraphs is far from complete, so browse through the manual and check with people around you for other possibilities.

eqn and **neqn** let you integrate mathematics into the text of a document, in an easy-to-learn language that closely resembles the way you would speak it aloud. For example, the **eqn** input

sum from i=0 to n x sub i ˜=˜ pi over 2

produces the output

$$\sum_{i=0}^{n} x_i ~=~ \frac{\pi}{2}$$

The program **tbl** provides an analogous service for preparing tabular material; it does all the computations necessary to align complicated columns with elements of varying widths.

refer prepares bibliographic citations from a data base, in whatever style is defined by the formatting package. It looks after all the details of numbering references in sequence, filling in page and volume numbers, getting the author’s initials and the journal name right, and so on.

spell and **typo** detect possible spelling mistakes in a document. **spell** works by comparing the words in your document to a dictionary, printing those that are not in the dictionary. It knows enough about English spelling to detect plurals and the like, so it does a very good job. **typo** looks for words which are ‘‘unusual’’, and prints those. Spelling mistakes tend to be more unusual, and thus show up early when the most unusual words are printed first.

grep looks through a set of files for lines that contain a particular text pattern (rather like the editor’s context search does, but on a bunch of files). For example,

grep ′ing$′ chap*

will find all lines that end with the letters **ing** in the files **chap***. (It is almost always a good practice to put single quotes around the pattern you’re searching for, in case it contains characters like * or $ that have a special meaning to the shell.) **grep** is often useful for finding out in which of a set of files the misspelled words detected by **spell** are actually located.

diff prints a list of the differences between two files, so you can compare two versions of something automatically (which certainly beats proofreading by hand).

wc counts the words, lines and characters in a set of files. **tr** translates characters into other characters; for example it will convert upper to lower case and vice versa. This translates upper into lower:

tr A−Z a−z <input >output

sort sorts files in a variety of ways; **cref** makes cross-references; **ptx** makes a permuted index (keyword-in-context listing). **sed** provides many of the editing facilities of **ed**, but can apply them to arbitrarily long inputs. **awk** provides the ability to do both pattern matching and numeric computations, and to conveniently process fields within lines. These programs are for more advanced users, and they are not limited to document preparation. Put them on your list of things to learn about.

Most of these programs are either independently documented (like **eqn** and **tbl**), or are sufficiently simple that the description in the *UNIX Programmer's Manual* is adequate explanation.

Hints for Preparing Documents

Most documents go through several versions (always more than you expected) before they are finally finished. Accordingly, you should do whatever possible to make the job of changing them easy.

First, when you do the purely mechanical operations of typing, type so that subsequent editing will be easy. Start each sentence on a new line. Make lines short, and break lines at natural places, such as after commas and semicolons, rather than randomly. Since most people change documents by rewriting phrases and adding, deleting and rearranging sentences, these precautions simplify any editing you have to do later.

Keep the individual files of a document down to modest size, perhaps ten to fifteen thousand characters. Larger files edit more slowly, and of course if you make a dumb mistake it's better to have clobbered a small file than a big one. Split into files at natural boundaries in the document, for the same reasons that you start each sentence on a new line.

The second aspect of making change easy is to not commit yourself to formatting details too early. One of the advantages of formatting packages like **–ms** is that they permit you to delay decisions to the last possible moment. Indeed, until a document is printed, it is not even decided whether it will be typeset or put on a line printer.

As a rule of thumb, for all but the most trivial jobs, you should type a document in terms of a set of requests like **.PP**, and then define them appropriately, either by using one of the canned packages (the better way) or by defining your own **nroff** and **troff** commands. As long as you have entered the text in some systematic way, it can always be cleaned up and re-formatted by a judicious combination of editing commands and request definitions.

IV. PROGRAMMING

There will be no attempt made to teach any of the programming languages available but a few words of advice are in order. One of the reasons why the UNIX system is a productive programming environment is that there is already a rich set of tools available, and facilities like pipes, I/O redirection, and the capabilities of the shell often make it possible to do a job by pasting together programs that already exist instead of writing from scratch.

The Shell

The pipe mechanism lets you fabricate quite complicated operations out of spare parts that already exist. For example, the first draft of the **spell** program was (roughly)

cat ...	*collect the files*
| tr ...	*put each word on a new line*
| tr ...	*delete punctuation, etc.*
| sort	*into dictionary order*
| uniq	*discard duplicates*
| comm	*print words in text but not in dictionary*

More pieces have been added subsequently, but this goes a long way for such a small effort.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines of each of a set of files, such as a book, you could laboriously type

```
ed
e chap1.1
1p
$p
e chap1.2
1p
$p
etc.
```

But you can do the job much more easily. One way is to type

```
ls chap* >temp
```

to get the list of filenames into a file. Then edit this file to make the necessary series of editing commands (using the global commands of **ed**), and write it into **script**. Now the command

```
ed <script
```

will produce the same output as the laborious hand typing. Alternately (and more easily), you can use the fact that the shell will perform loops, repeating a set of commands over and over again for a set of arguments:

```
for i in chap*
do
     ed $i <script
done
```

This sets the shell variable **i** to each file name in turn, then does the command. You can type this command at the terminal, or put it in a file for later execution.

Programming the Shell

An option often overlooked by newcomers is that the shell is itself a programming language, with variables, control flow (**if-else**, **while**, **for**, **case**), subroutines, and interrupt handling. Since there are many building-block programs, you can sometimes avoid writing a new program merely by piecing together some of the building blocks with shell command files.

We will not go into any details here; examples and rules can be found in *An Introduction to the UNIX Shell*, by S. R. Bourne.

Programming in C

If you are undertaking anything substantial, C is the only reasonable choice of programming language: everything in the UNIX system is tuned to it. The system itself is written in C, as are most of the programs that run on it. It is also a easy language to use once you get started. C is introduced and fully described in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Several sections of the manual describe the system interfaces, that is, how you do I/O and similar functions. Read *UNIX Programming* for more complicated things.

Most input and output in C is best handled with the standard I/O library, which provides a set of I/O functions that exist in compatible form on most machines that have C compilers. In general, it's wisest to confine the system interactions in a program to the facilities provided by this library.

C programs that don't depend too much on special features of UNIX (such as pipes) can be moved to other computers that have C compilers. The list of such machines grows daily; in addition to the original PDP-11, it currently includes at least Honeywell 6000, IBM 370, Interdata 8/32, Data General Nova and Eclipse, HP 2100, Harris /7, VAX 11/780, SEL 86, and Zilog Z80. Calls to the standard I/O library will work on all of these machines.

There are a number of supporting programs that go with C. **lint** checks C programs for potential portability problems, and detects errors such as mismatched argument types and uninitialized variables.

For larger programs (anything whose source is on more than one file) **make** allows you to specify the dependencies among the source files and the processing steps needed to make a new version; it then checks the times that the pieces were last changed and does the minimal amount of recompiling to create a consistent updated version.

The debugger **adb** is useful for digging through the dead bodies of C programs, but is rather hard to learn to use effectively. The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

The C compiler provides a limited instrumentation service, so you can find out where programs spend their time and what parts are worth optimizing. Compile the routines with the **−p** option; after the test run, use **prof** to print an execution profile. The command **time** will give you the gross run-time statistics of a program, but they are not super accurate or reproducible.

Other Languages

If you *have* to use Fortran, there are two possibilities. You might consider Ratfor, which gives you the decent control structures and free-form input that characterize C, yet lets you write code that is still portable to other environments. Bear in mind that UNIX Fortran tends to produce large and relatively slow-running programs. Furthermore, supporting software like **adb**, **prof**, etc., are all virtually useless with Fortran programs. There may also be a Fortran 77 compiler on your system. If so, this is a viable alternative to Ratfor, and has the non-trivial advantage that it is compatible with C and related programs. (The Ratfor processor and C tools can be used with Fortran 77 too.)

If your application requires you to translate a language into a set of actions or another language, you are in effect building a compiler, though probably a small one. In that case, you should be using the **yacc** compiler-compiler, which helps you develop a compiler quickly. The **lex** lexical analyzer generator does the same job for the simpler languages that can be expressed as regular expressions. It can be used by itself, or as a front end to recognize inputs for a **yacc**-based program. Both **yacc** and **lex** require some sophistication to use, but the initial effort of learning them can be repaid many times over in programs that are easy to change later on.

Most UNIX systems also make available other languages, such as Algol 68, APL, Basic, Lisp, Pascal, and Snobol. Whether these are useful depends largely on the local environment: if someone cares about the language and has worked on it, it may be in good shape. If not, the odds are strong that it will be more trouble than it's worth.

V. UNIX READING LIST

General:

K. L. Thompson and D. M. Ritchie, *The UNIX Programmer's Manual,* Bell Laboratories, 1978. Lists commands, system routines and interfaces, file formats, and some of the maintenance procedures. You can't live without this, although you will probably only need to read section 1.

Documents for Use with the UNIX Time-sharing System. Volume 2 of the Programmer's Manual. This contains more extensive descriptions of major commands, and tutorials and reference manuals. All of the papers listed below are in it, as are descriptions of most of the programs mentioned above.

D. M. Ritchie and K. L. Thompson, ''The UNIX Time-sharing System,'' CACM, July 1974. An overview of the system, for people interested in operating systems. Worth reading by anyone who programs. Contains a remarkable number of one-sentence observations on how to do things right.

The Bell System Technical Journal (BSTJ) Special Issue on UNIX, July/August, 1978, contains many papers describing recent developments, and some retrospective material.

The 2nd International Conference on Software Engineering (October, 1976) contains several papers describing the use of the Programmer's Workbench (PWB) version of UNIX.

Document Preparation:

B. W. Kernighan, ''A Tutorial Introduction to the UNIX Text Editor'' and ''Advanced Editing on UNIX,'' Bell Laboratories, 1978. Beginners need the introduction; the advanced material will help you get the most out of the editor.

M. E. Lesk, ''Typing Documents on UNIX,'' Bell Laboratories, 1978. Describes the **–ms** macro package, which isolates the novice from the vagaries of **nroff** and **troff**, and takes care of most formatting situations. If this specific package isn't available on your system, something similar probably is. The most likely alternative is the PWB/UNIX macro package **–mm**; see your local guru if you use PWB/UNIX.

B. W. Kernighan and L. L. Cherry, ''A System for Typesetting Mathematics,'' Bell Laboratories Computing Science Tech. Rep. 17.

M. E. Lesk, ''Tbl — A Program to Format Tables,'' Bell Laboratories CSTR 49, 1976.

J. F. Ossanna, Jr., ''NROFF/TROFF User's Manual,'' Bell Laboratories CSTR 54, 1976. **troff** is the basic formatter used by **–ms**, **eqn** and **tbl**. The reference manual is indispensable if you are going to write or maintain these or similar programs. But start with:

B. W. Kernighan, ''A TROFF Tutorial,'' Bell Laboratories, 1976. An attempt to unravel the intricacies of **troff**.

Programming:

B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978. Contains a tutorial introduction, complete discussions of all language features, and the reference manual.

B. W. Kernighan and D. M. Ritchie, ''UNIX Programming,'' Bell Laboratories, 1978. Describes how to interface with the system from C programs: I/O calls, signals, processes.

S. R. Bourne, ''An Introduction to the UNIX Shell,'' Bell Laboratories, 1978. An introduction and reference manual for the Version 7 shell. Mandatory reading if you intend to make effective use of the programming power of this shell.

S. C. Johnson, ''Yacc — Yet Another Compiler-Compiler,'' Bell Laboratories CSTR 32, 1978.

M. E. Lesk, ''Lex — A Lexical Analyzer Generator,'' Bell Laboratories CSTR 39, 1975.

S. C. Johnson, ''Lint, a C Program Checker,'' Bell Laboratories CSTR 65, 1977.

S. I. Feldman, ''MAKE — A Program for Maintaining Computer Programs,'' Bell Laboratories CSTR 57, 1977.

J. F. Maranzano and S. R. Bourne, ''A Tutorial Introduction to ADB,'' Bell Laboratories CSTR 62, 1977. An introduction to a powerful but complex debugging tool.

S. I. Feldman and P. J. Weinberger, ''A Portable Fortran 77 Compiler,'' Bell Laboratories, 1978. A full Fortran 77 for UNIX systems.

A Tutorial Introduction to the UNIX Text Editor

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Almost all text input on the UNIX† operating system is done with the text-editor *ed*. This memorandum is a tutorial guide to help beginners get started with text editing.

Although it does not cover everything, it does discuss enough for most users' day-to-day needs. This includes printing, appending, changing, deleting, moving and inserting entire lines of text; reading and writing files; context searching and line addressing; the substitute command; the global commands; and the use of special characters for advanced editing.

September 21, 1978

A Tutorial Introduction to the UNIX Text Editor

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

Ed is a ‘‘text editor’’, that is, an interactive program for creating and modifying ‘‘text’’, using directions provided by a user at a terminal. The text is often a document like this one, or a program or perhaps data for a program.

This introduction is meant to simplify learning *ed.* The recommended way to learn *ed* is to read this document, simultaneously using *ed* to follow the examples, then to read the description in section I of the *UNIX Programmer's Manual,* all the while experimenting with *ed.* (Solicitation of advice from experienced users is also useful.)

Do the exercises! They cover material not completely discussed in the actual text. An appendix summarizes the commands.

Disclaimer

This is an introduction and a tutorial. For this reason, no attempt is made to cover more than a part of the facilities that *ed* offers (although this fraction includes the most useful and frequently used parts). When you have mastered the Tutorial, try *Advanced Editing on UNIX.* Also, there is not enough space to explain basic UNIX procedures. We will assume that you know how to log on to UNIX, and that you have at least a vague understanding of what a file is. For more on that, read *UNIX for Beginners.*

You must also know what character to type as the end-of-line on your particular terminal. This character is the RETURN key on most terminals. Throughout, we will refer to this character, whatever it is, as RETURN.

Getting Started

We'll assume that you have logged in to your system and it has just printed the prompt character, usually either a **$** or a **%**. The easiest way to get *ed* is to type

```
ed        (followed by a return)
```

You are now ready to go – *ed* is waiting for you to tell it what to do.

Creating Text – the Append command ‘‘a’’

As your first problem, suppose you want to create some text starting from scratch. Perhaps you are typing the very first draft of a paper; clearly it will have to start somewhere, and undergo modifications later. This section will show how to get some text in, just to get started. Later we'll talk about how to change it.

When *ed* is first started, it is rather like working with a blank piece of paper – there is no text or information present. This must be supplied by the person using *ed;* it is usually done by typing in the text, or by reading it into *ed* from a file. We will start by typing in some text, and return shortly to how to read files.

First a bit of terminology. In *ed* jargon, the text being worked on is said to be ‘‘kept in a buffer.’’ Think of the buffer as a work space, if you like, or simply as the information that you are going to be editing. In effect the buffer is like the piece of paper, on which we will write things, then change some of them, and finally file the whole thing away for another day.

The user tells *ed* what to do to his text by typing instructions called ‘‘commands.’’ Most commands consist of a single letter, which must be typed in lower case. Each command is typed on a separate line. (Sometimes the command is preceded by information about what line or lines of text are to be affected – we will discuss these shortly.) *Ed* makes no response to most commands – there is no prompting or typing of messages like ‘‘ready’’. (This silence is preferred by experienced users, but sometimes a hangup for beginners.)

The first command is *append,* written as the letter

```
a
```

all by itself. It means ‘‘append (or add) text lines to the buffer, as I type them in.’’ Appending is rather like writing fresh material on a piece of paper.

So to enter lines of text into the buffer, just type an **a** followed by a RETURN, followed by the lines of text you want, like this:

```
a
Now is the time
for all good men
to come to the aid of their party.
.
```

The only way to stop appending is to type a line that contains only a period. The ''.'' is used to tell *ed* that you have finished appending. (Even experienced users forget that terminating ''.'' sometimes. If *ed* seems to be ignoring you, type an extra line with just ''.'' on it. You may then find you've added some garbage lines to your text, which you'll have to take out later.)

After the append command has been done, the buffer will contain the three lines

```
Now is the time
for all good men
to come to the aid of their party.
```

The ''**a**'' and ''**.**'' aren't there, because they are not text.

To add more text to what you already have, just issue another **a** command, and continue typing.

Error Messages – ''?''

If at any time you make an error in the commands you type to *ed,* it will tell you by typing

```
?
```

This is about as cryptic as it can be, but with practice, you can usually figure out how you goofed.

Writing text out as a file – the Write command ''w''

It's likely that you'll want to save your text for later use. To write out the contents of the buffer onto a file, use the *write* command

```
w
```

followed by the filename you want to write on. This will copy the buffer's contents onto the specified file (destroying any previous information on the file). To save the text on a file named **junk**, for example, type

```
w junk
```

Leave a space between **w** and the file name. *Ed* will respond by printing the number of characters it wrote out. In this case, *ed* would respond with

```
68
```

(Remember that blanks and the return character at the end of each line are included in the character count.) Writing a file just makes a copy of the text – the buffer's contents are not disturbed, so you can go on adding lines to it. This is an important point. *Ed* at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a **w** command. (Writing out the text onto a file from time to time as it is being created is a good idea, since if the system crashes or if you make some horrible mistake, you will lose all the text in the buffer but any text that was written onto a file is relatively safe.)

Leaving ed – the Quit command ''q''

To terminate a session with *ed*, save the text you're working on by writing it onto a file using the **w** command, and then type the command

```
q
```

which stands for *quit*. The system will respond with the prompt character (**$** or **%**). At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting.†

Exercise 1:

Enter *ed* and create some text using

```
a
. . . text . . .
.
```

Write it out using **w**. Then leave *ed* with the **q** command, and print the file, to see that everything worked. (To print a file, say

```
pr filename
```

or

```
cat filename
```

in response to the prompt character. Try both.)

Reading text from a file – the Edit command ''e''

A common way to get text into the buffer is to read it from a file in the file system. This is what you do to edit text that you saved with the **w** command in a previous session. The *edit* command **e** fetches the entire contents of a file into the buffer. So if you had saved the three lines ''Now is the time'', etc., with a **w** command in an earlier session, the *ed* command

```
e junk
```

would fetch the entire contents of the file **junk** into the buffer, and respond

```
68
```

which is the number of characters in **junk**. *If anything was already in the buffer, it is deleted first.*

If you use the **e** command to read a file into the buffer, then you need not use a file name after a subsequent **w** command; *ed* remembers the last file name

† Actually, *ed* will print **?** if you try to quit without writing. At that point, write if you want; if not, another **q** will get you out regardless.

used in an **e** command, and **w** will write on this file. Thus a good way to operate is

```
ed
e file
[editing session]
w
q
```

This way, you can simply say **w** from time to time, and be secure in the knowledge that if you got the file name right at the beginning, you are writing into the proper file each time.

You can find out at any time what file name *ed* is remembering by typing the *file* command **f**. In this example, if you typed

```
f
```

ed would reply

```
junk
```

Reading text from a file – the Read command ''r''

Sometimes you want to read a file into the buffer without destroying anything that is already there. This is done by the *read* command **r**. The command

```
r junk
```

will read the file **junk** into the buffer; it adds it to the end of whatever is already in the buffer. So if you do a read after an edit:

```
e junk
r junk
```

the buffer will contain *two* copies of the text (six lines).

```
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
```

Like the **w** and **e** commands, **r** prints the number of characters read in, after the reading operation is complete.

Generally speaking, **r** is much less used than **e**.

Exercise 2:

Experiment with the **e** command – try reading and printing various files. You may get an error **?name**, where **name** is the name of a file; this means that the file doesn't exist, typically because you spelled the file name wrong, or perhaps that you are not allowed to read or write it. Try alternately reading and appending to see that they work similarly. Verify that

```
ed filename
```

is exactly equivalent to

```
ed
e filename
```

What does

```
f filename
```

do?

Printing the contents of the buffer – the Print command ''p''

To *print* or list the contents of the buffer (or parts of it) on the terminal, use the print command

```
p
```

The way this is done is as follows. Specify the lines where you want printing to begin and where you want it to end, separated by a comma, and followed by the letter **p**. Thus to print the first two lines of the buffer, for example, (that is, lines 1 through 2) say

```
1,2p      (starting line=1, ending line=2 p)
```

Ed will respond with

```
Now is the time
for all good men
```

Suppose you want to print *all* the lines in the buffer. You could use **1,3p** as above if you knew there were exactly 3 lines in the buffer. But in general, you don't know how many there are, so what do you use for the ending line number? *Ed* provides a shorthand symbol for ''line number of last line in buffer'' – the dollar sign **$**. Use it this way:

```
1,$p
```

This will print *all* the lines in the buffer (line 1 to last line.) If you want to stop the printing before it is finished, push the DEL or Delete key; *ed* will type

```
?
```

and wait for the next command.

To print the *last* line of the buffer, you could use

```
$,$p
```

but *ed* lets you abbreviate this to

```
$p
```

You can print any single line by typing the line number followed by a **p**. Thus

```
1p
```

produces the response

```
Now is the time
```

which is the first line of the buffer.

In fact, *ed* lets you abbreviate even further: you can print any single line by typing *just* the line number – no need to type the letter **p**. So if you say

$

ed will print the last line of the buffer.

You can also use **$** in combinations like

$–1,$p

which prints the last two lines of the buffer. This helps when you want to see how far you got in typing.

Exercise 3:

As before, create some text using the **a** command and experiment with the **p** command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that attempts to print a buffer in reverse order by saying

3,1p

don't work.

The current line – ''Dot'' or ''.''

Suppose your buffer still contains the six lines as above, that you have just typed

1,3p

and *ed* has printed the three lines for you. Try typing just

p (no line numbers)

This will print

to come to the aid of their party.

which is the third line of the buffer. In fact it is the last (most recent) line that you have done anything with. (You just printed it!) You can repeat this **p** command without line numbers, and it will continue to print line 3.

The reason is that *ed* maintains a record of the last line that you did anything to (in this case, line 3, which you just printed) so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol

. (pronounced ''dot'').

Dot is a line number in the same way that **$** is; it means exactly ''the current line'', or loosely, ''the line you most recently did something to.'' You can use it in several ways – one possibility is to say

.,$p

This will print all the lines from (including) the current line to the end of the buffer. In our example these are lines 3 through 6.

Some commands change the value of dot, while others do not. The **p** command sets dot to the number of the last line printed; the last command will set both **.** and **$** to 6.

Dot is most useful when used in combinations like this one:

.+1 (or equivalently, .+1p)

This means ''print the next line'' and is a handy way to step slowly through a buffer. You can also say

.–1 (or .–1p)

which means ''print the line *before* the current line.'' This enables you to go backwards if you wish. Another useful one is something like

.–3,.–1p

which prints the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by typing

.=

Ed will respond by printing the value of dot.

Let's summarize some things about the **p** command and dot. Essentially **p** can be preceded by 0, 1, or 2 line numbers. If there is no line number given, it prints the ''current line'', the line that dot refers to. If there is one line number given (with or without the letter **p**), it prints that line (and dot is set there); and if there are two line numbers, it prints all the lines in that range (and sets dot to the last line printed.) If two line numbers are specified the first can't be bigger than the second (see Exercise 2.)

Typing a single return will cause printing of the next line – it's equivalent to **.+1p**. Try it. Try typing a –; you will find that it's equivalent to **.–1p**.

Deleting lines: the ''d'' command

Suppose you want to get rid of the three extra lines in the buffer. This is done by the *delete* command

d

Except that **d** deletes lines instead of printing them, its action is similar to that of **p**. The lines to be deleted are specified for **d** exactly as they are for **p**:

starting line, ending line d

Thus the command

4,$d

deletes lines 4 through the end. There are now three lines left, as you can check by using

1,$p

And notice that **$** now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to **$**.

Exercise 4:

Experiment with **a**, **e**, **r**, **w**, **p** and **d** until you are sure that you know what they do, and until you understand how dot, **$**, and line numbers are used.

If you are adventurous, try using line numbers with **a**, **r** and **w** as well. You will find that **a** will append lines *after* the line number that you specify (rather than after dot); that **r** reads a file in *after* the line number you specify (not necessarily at the end of the buffer); and that **w** will write out exactly the lines you specify, not necessarily the whole buffer. These variations are sometimes handy. For instance you can insert a file at the beginning of a buffer by saying

```
0r filename
```

and you can enter lines at the beginning of the buffer by saying

```
0a
. . . text . . .
.
```

Notice that **.w** is *very* different from

```
.
w
```

Modifying text: the Substitute command "s"

We are now ready to try one of the most important of all commands – the substitute command

```
s
```

This is the command that is used to change individual words or letters within a line or group of lines. It is what you use, for example, for correcting spelling mistakes and typing errors.

Suppose that by a typing error, line 1 says

```
Now is th time
```

– the *e* has been left off *the*. You can use **s** to fix this up as follows:

```
1s/th/the/
```

This says: "in line 1, substitute for the characters *th* the characters *the*." To verify that it works (*ed* will not print the result automatically) say

```
p
```

and get

```
Now is the time
```

which is what you wanted. Notice that dot must have been set to the line where the substitution took place, since the **p** command printed that line. Dot is always set this way with the **s** command.

The general way to use the substitute command is

starting-line, ending-line s/*change this*/*to this*/

Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between *starting-line* and *ending-line*. Only the first occurrence on each line is changed, however. If you want to change *every* occurrence, see Exercise 5. The rules for line numbers are the same as those for **p**, except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution took place, dot is *not* changed. This causes an error **?** as a warning.)

Thus you can say

```
1,$s/speling/spelling/
```

and correct the first spelling mistake on each line in the text. (This is useful for people who are consistent misspellers!)

If no line numbers are given, the **s** command assumes we mean "make the substitution on line dot", so it changes things only on the current line. This leads to the very common sequence

```
s/something/something else/p
```

which makes some correction on the current line, and then prints it, to make sure it worked out right. If it didn't, you can try again. (Notice that there is a **p** on the same line as the **s** command. With few exceptions, **p** can follow any command; no other multi-command lines are legal.)

It's also legal to say

```
s/ . . . //
```

which means "change the first string of characters to "*nothing*", i.e., remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if you had

```
Nowxx is the time
```

you can say

```
s/xx//p
```

to get

```
Now is the time
```

Notice that // (two adjacent slashes) means "no characters", not a blank. There *is* a difference! (See below for another meaning of //.)

Exercise 5:

Experiment with the substitute command. See what happens if you substitute for some word on a line with several occurrences of that word. For example, do this:

```
a
the other side of the coin
.
s/the/on the/p
```

You will get

on the other side of the coin

A substitute command changes only the first occurrence of the first string. You can change all occurrences by adding a **g** (for ‘‘global’’) to the **s** command, like this:

s/ . . . / . . . /gp

Try other characters instead of slashes to delimit the two sets of characters in the **s** command – anything should work except blanks or tabs.

(If you get funny results using any of the characters

^ . $ [* \ &

read the section on ‘‘Special Characters’’.)

Context searching – ‘‘/ . . . /’’

With the substitute command mastered, you can move on to another highly important idea of *ed* – context searching.

Suppose you have the original three line text in the buffer:

Now is the time
for all good men
to come to the aid of their party.

Suppose you want to find the line that contains *their* so you can change it to *the*. Now with only three lines in the buffer, it's pretty easy to keep track of what line the word *their* is on. But if the buffer contained several hundred lines, and you'd been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

The way to say ‘‘search for a line that contains this particular string of characters’’ is to type

/string of characters we want to find/

For example, the *ed* command

/their/

is a context search which is sufficient to find the desired line – it will locate the next occurrence of the characters between slashes (‘‘their’’). It also sets dot to that line and prints the line for verification:

to come to the aid of their party.

‘‘Next occurrence’’ means that *ed* starts looking for the string at line **.+1**, searches to the end of the buffer, then continues at line 1 and searches to line dot. (That is, the search ‘‘wraps around’’ from **$** to 1.) It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can't be found in any line, *ed* types the error message

?

Otherwise it prints the line it found.

You can do both the search for the desired line *and* a substitution all at once, like this:

/their/s/their/the/p

which will yield

to come to the aid of the party.

There were three parts to that last command: context search for the desired line, make the substitution, print the line.

The expression **/their/** is a context search expression. In their simplest form, all context search expressions are like this – a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line, or as line numbers for some other command, like **s**. They were used both ways in the examples above.

Suppose the buffer contains the three familiar lines

Now is the time
for all good men
to come to the aid of their party.

Then the *ed* line numbers

/Now/+1
/good/
/party/−1

are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, you could say

/Now/+1s/good/bad/

or

/good/s/good/bad/

or

/party/−1s/good/bad/

The choice is dictated only by convenience. You could print all three lines by, for instance

/Now/,/party/p

or

/Now/,/Now/+2p

or by any number of similar combinations. The first one of these might be better if you don't know how many lines are involved. (Of course, if there were only three lines in the buffer, you'd use

1,$p

but not if there were several hundred.)

The basic rule is: a context search expression is *the same as* a line number, so it can be used wherever a line number is needed.

Exercise 6:

Experiment with context searching. Try a body of text with several occurrences of the same string of characters, and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print and delete commands. (They can also be used with **r**, **w**, and **a**.)

Try context searching using **?text?** instead of **/text/**. This scans lines in the buffer in reverse order rather than normal. This is sometimes useful if you go too far while looking for some string of characters – it's an easy way to back up.

(If you get funny results with any of the characters

```
^   .   $   [   *   \   &
```

read the section on ''Special Characters''.)

Ed provides a shorthand for repeating a context search for the same string. For example, the *ed* line number

```
/string/
```

will find the next occurrence of **string**. It often happens that this is not the desired line, so the search must be repeated. This can be done by typing merely

```
//
```

This shorthand stands for ''the most recently used context search expression.'' It can also be used as the first string of the substitute command, as in

```
/string1/s//string2/
```

which will find the next occurrence of **string1** and replace it by **string2**. This can save a lot of typing. Similarly

```
??
```

means ''scan backwards for the same expression.''

Change and Insert – ''c'' and ''i''

This section discusses the *change* command

```
c
```

which is used to change or replace a group of one or more lines, and the *insert* command

```
i
```

which is used for inserting a group of one or more lines.

''Change'', written as

```
c
```

is used to replace a number of lines with different lines, which are typed in at the terminal. For example, to change lines **.+1** through **$** to something else, type

.+1,$c
. . . *type the lines of text you want here* . . .
.

The lines you type between the **c** command and the **.** will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines which have errors in them.

If only one line is specified in the **c** command, then just that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of **.** to end the input – this works just like the **.** in the append command and must appear by itself on a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed in.

''Insert'' is similar to append – for instance

/string/i
. . . *type the lines to be inserted here* . . .
.

will insert the given text *before* the next line that contains ''string''. The text between **i** and **.** is *inserted before* the specified line. If no line number is specified dot is used. Dot is set to the last line inserted.

Exercise 7:

''Change'' is rather like a combination of delete followed by insert. Experiment to verify that

start, end d
i
. . . *text* . . .
.

is almost the same as

start, end c
. . . *text* . . .
.

These are not *precisely* the same if line **$** gets deleted. Check this out. What is dot?

Experiment with **a** and **i**, to see that they are similar, but not the same. You will observe that

line-number a
. . . *text* . . .
.

appends *after* the given line, while

line-number i
. . . *text* . . .
.

inserts *before* it. Observe that if no line number is given, **i** inserts before line dot, while **a** appends after line dot.

Moving text around: the ‘‘m’’ command

The move command **m** is used for cutting and pasting – it lets you move a group of lines from one place to another in the buffer. Suppose you want to put the first three lines of the buffer at the end instead. You could do it by saying:

```
1,3w temp
$r temp
1,3d
```

(Do you see why?) but you can do it a lot easier with the **m** command:

```
1,3m$
```

The general case is

start line, end line m *after this line*

Notice that there is a third line to be specified – the place where the moved stuff gets put. Of course the lines to be moved can be specified by context searches; if you had

```
First paragraph
. . .
end of first paragraph.
Second paragraph
. . .
end of second paragraph.
```

you could reverse the two paragraphs like this:

```
/Second/,/end of second/m/First/−1
```

Notice the **−1**: the moved text goes *after* the line mentioned. Dot gets set to the last line moved.

The global commands ‘‘g’’ and ‘‘v’’

The *global* command **g** is used to execute one or more *ed* commands on all those lines in the buffer that match some specified string. For example

```
g/peling/p
```

prints all lines that contain **peling**. More usefully,

```
g/peling/s//pelling/gp
```

makes the substitution everywhere on the line, then prints each corrected line. Compare this to

```
1,$s/peling/pelling/gp
```

which only prints the last line substituted. Another subtle difference is that the **g** command does not give a **?** if **peling** is not found where the **s** command will.

There may be several commands (including **a**, **c**, **i**, **r**, **w**, but not **g**); in that case, every line except the last must end with a backslash \:

```
g/xxx/.−1s/abc/def/\
.+2s/ghi/jkl/\
.−2,.p
```

makes changes in the lines before and after each line that contains **xxx**, then prints all three lines.

The **v** command is the same as **g**, except that the commands are executed on every line that does *not* match the string following **v**:

```
v/ /d
```

deletes every line that does not contain a blank.

Special Characters

You may have noticed that things just don’t work right when you used some characters like **.**, *, **$**, and others in context searches and the substitute command. The reason is rather complex, although the cure is simple. Basically, *ed* treats these characters as special, with special meanings. For instance, *in a context search or the first string of the substitute command only,* **.** means ‘‘any character,’’ not a period, so

```
/x.y/
```

means ‘‘a line with an **x**, *any character,* and a **y**,’’ *not* just ‘‘a line with an **x**, a period, and a **y**.’’ A complete list of the special characters that can cause trouble is the following:

```
^    .    $    [    *    \
```

Warning: The backslash character \ is special to *ed.* For safety’s sake, avoid it where possible. If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus

```
s/\\\.\*/backslash dot star/
```

will change **\.*** into ‘‘backslash dot star’’.

Here is a hurried synopsis of the other special characters. First, the circumflex ^ signifies the beginning of a line. Thus

```
/^string/
```

finds **string** only if it is at the beginning of a line: it will find

```
string
```

but not

```
the string...
```

The dollar-sign **$** is just the opposite of the circumflex; it means the end of a line:

```
/string$/
```

will only find an occurrence of **string** that is at the end of some line. This implies, of course, that

```
/^string$/
```

will find only a line that contains just **string**, and

/^.$/

finds a line containing exactly one character.

The character **.**, as we mentioned above, matches anything;

/x.y/

matches any of

x+y
x−y
x y
x.y

This is useful in conjunction with ∗, which is a repetition character; **a**∗ is a shorthand for ''any number of **a**'s,'' so **.**∗ matches any number of anythings. This is used like this:

s/.∗/stuff/

which changes an entire line, or

s/.∗,//

which deletes all characters in the line up to and including the last comma. (Since **.**∗ finds the longest possible match, this goes up to the last comma.)

[is used with **]** to form ''character classes''; for example,

/[0123456789]/

matches any single digit – any one of the characters inside the braces will cause a match. This can be abbreviated to **[0−9]**.

Finally, the **&** is another shorthand character – it is used only on the right-hand part of a substitute command where it means ''whatever was matched on the left-hand side''. It is used to save typing. Suppose the current line contained

Now is the time

and you wanted to put parentheses around it. You could just retype the line, but this is tedious. Or you could say

s/^/(/
s/$/)/

using your knowledge of ^ and **$**. But the easiest way uses the **&**:

s/.∗/(&)/

This says ''match the whole line, and replace it by itself surrounded by parentheses.'' The **&** can be used several times in a line; consider using

s/.∗/&? &!!/

to produce

Now is the time? Now is the time!!

You don't have to match the whole line, of course: if the buffer contains

the end of the world

you could type

/world/s//& is at hand/

to produce

the end of the world is at hand

Observe this expression carefully, for it illustrates how to take advantage of *ed* to save typing. The string **/world/** found the desired line; the shorthand **//** found the same word in the line; and the **&** saves you from typing it again.

The **&** is a special character only within the replacement text of a substitute command, and has no special meaning elsewhere. You can turn off the special meaning of **&** by preceding it with a \:

s/ampersand/\&/

will convert the word ''ampersand'' into the literal symbol **&** in the current line.

Summary of Commands and Line Numbers

The general form of *ed* commands is the command name, perhaps preceded by one or two line numbers, and, in the case of **e**, **r**, and **w**, followed by a file name. Only one command is allowed per line, but a **p** command may follow any other command (except for **e**, **r**, **w** and **q**).

a: Append, that is, add lines to the buffer (at line dot, unless a different line is specified). Appending continues until **.** is typed on a new line. Dot is set to the last line appended.

c: Change the specified lines to the new text which follows. The new lines are terminated by a **.**, as with **a**. If no lines are specified, replace line dot. Dot is set to last line changed.

d: Delete the lines specified. If none are specified, delete line dot. Dot is set to the first undeleted line, unless **$** is deleted, in which case dot is set to **$**.

e: Edit new file. Any previous contents of the buffer are thrown away, so issue a **w** beforehand.

f: Print remembered filename. If a name follows **f** the remembered name will be set to it.

g: The command

g/---/commands

will execute the commands on those lines that contain **---**, which can be any context search expression.

i: Insert lines before specified line (or dot) until a **.** is typed on a new line. Dot is set to last line inserted.

m: Move lines specified to after the line named after **m**. Dot is set to the last line moved.

p: Print specified lines. If none specified, print line dot. A single line number is equivalent to *line-number* **p**. A single return prints **.+1**, the next line.

q: Quit *ed*. Wipes out all text in buffer if you give it twice in a row without first giving a **w** command.

r: Read a file into buffer (at end unless specified elsewhere.) Dot set to last line read.

s: The command

    s/string1/string2/

substitutes the characters **string1** into **string2** in the specified lines. If no lines are specified, make the substitution in line dot. Dot is set to last line in which a substitution took place, which means that if no substitution took place, dot is not changed. **s** changes only the first occurrence of **string1** on a line; to change all of them, type a **g** after the final slash.

v: The command

    v/---/commands

executes **commands** on those lines that *do not* contain **---**.

w: Write out buffer onto a file. Dot is not changed.

.=: Print value of dot. (**=** by itself prints the value of **$**.)

!: The line

    !command-line

causes **command-line** to be executed as a UNIX command.

/-----/: Context search. Search for next line which contains this string of characters. Print it. Dot is set to the line where string was found. Search starts at **.+1**, wraps around from **$** to 1, and continues to dot, if necessary.

?-----?: Context search in reverse direction. Start search at **.−1**, scan to 1, wrap around to **$**.

Advanced Editing on UNIX

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is meant to help secretaries, typists and programmers to make effective use of the UNIX† facilities for preparing and editing text. It provides explanations and examples of

- special characters, line addressing and global commands in the editor **ed**;
- commands for ''cut and paste'' operations on files and parts of files, including the **mv**, **cp**, **cat** and **rm** commands, and the **r**, **w**, **m** and **t** commands of the editor;
- editing scripts and editor-based programs like **grep** and **sed**.

Although the treatment is aimed at non-programmers, new users with any background should find helpful hints on how to get their jobs done more easily.

August 4, 1978

†UNIX is a Trademark of Bell Laboratories.

Advanced Editing on UNIX

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

Although UNIX† provides remarkably effective tools for text editing, that by itself is no guarantee that everyone will automatically make the most effective use of them. In particular, people who are not computer specialists — typists, secretaries, casual users — often use the system less effectively than they might.

This document is intended as a sequel to *A Tutorial Introduction to the UNIX Text Editor* [1], providing explanations and examples of how to edit with less effort. (You should also be familiar with the material in *UNIX For Beginners* [2].) Further information on all commands discussed here can be found in *The UNIX Programmer's Manual* [3].

Examples are based on observations of users and the difficulties they encounter. Topics covered include special characters in searches and substitute commands, line addressing, the global commands, and line moving and copying. There are also brief discussions of effective use of related tools, like those for file manipulation, and those based on **ed**, like **grep** and **sed**.

A word of caution. There is only one way to learn to use something, and that is to *use* it. Reading a description is no substitute for trying something. A paper like this one should give you ideas about what to try, but until you actually try something, you will not learn it.

2. SPECIAL CHARACTERS

The editor **ed** is the primary interface to the system for many people, so it is worthwhile to know how to get the most out of **ed** for the least effort.

The next few sections will discuss shortcuts and labor-saving devices. Not all of these will be instantly useful to any one person, of course, but a few will be, and the others should give you ideas to store away for future use. And as always, until you try these things, they will remain theoretical knowledge, not something you have confidence in.

†UNIX is a Trademark of Bell Laboratories.

The List command 'l'

ed provides two commands for printing the contents of the lines you're editing. Most people are familiar with **p**, in combinations like

```
1,$p
```

to print all the lines you're editing, or

```
s/abc/def/p
```

to change 'abc' to 'def' on the current line. Less familiar is the *list* command **l** (the letter '*l*'), which gives slightly more information than **p**. In particular, **l** makes visible characters that are normally invisible, such as tabs and backspaces. If you list a line that contains some of these, **l** will print each tab as ≻ and each backspace as ≺. This makes it much easier to correct the sort of typing mistake that inserts extra spaces adjacent to tabs, or inserts a backspace followed by a space.

The **l** command also 'folds' long lines for printing — any line that exceeds 72 characters is printed on multiple lines; each printed line except the last is terminated by a backslash ****, so you can tell it was folded. This is useful for printing long lines on short terminals.

Occasionally the **l** command will print in a line a string of numbers preceded by a backslash, such as \07 or \16. These combinations are used to make visible characters that normally don't print, like form feed or vertical tab or bell. Each such combination is a single character. When you see such characters, be wary — they may have surprising meanings when printed on some terminals. Often their presence means that your finger slipped while you were typing; you almost never want them.

The Substitute Command 's'

Most of the next few sections will be taken up with a discussion of the substitute command **s**. Since this is the command for changing the contents of individual lines, it probably has the most complexity of any **ed** command, and the most potential for effective use.

As the simplest place to begin, recall the meaning of a trailing **g** after a substitute command.

With

    s/this/that/

and

    s/this/that/g

the first one replaces the *first* 'this' on the line with 'that'. If there is more than one 'this' on the line, the second form with the trailing **g** changes *all* of them.

Either form of the **s** command can be followed by **p** or **l** to 'print' or 'list' (as described in the previous section) the contents of the line:

    s/this/that/p
    s/this/that/l
    s/this/that/gp
    s/this/that/gl

are all legal, and mean slightly different things. Make sure you know what the differences are.

Of course, any **s** command can be preceded by one or two 'line numbers' to specify that the substitution is to take place on a group of lines. Thus

    1,$s/mispell/misspell/

changes the *first* occurrence of 'mispell' to 'misspell' on every line of the file. But

    1,$s/mispell/misspell/g

changes *every* occurrence in every line (and this is more likely to be what you wanted in this particular case).

You should also notice that if you add a **p** or **l** to the end of any of these substitute commands, only the last line that got changed will be printed, not all the lines. We will talk later about how to print all the lines that were modified.

The Undo Command 'u'

Occasionally you will make a substitution in a line, only to realize too late that it was a ghastly mistake. The 'undo' command **u** lets you 'undo' the last substitution: the last line that was substituted can be restored to its previous state by typing the command

    u

The Metacharacter '.'

As you have undoubtedly noticed when you use **ed**, certain characters have unexpected meanings when they occur in the left side of a substitute command, or in a search for a particular line. In the next several sections, we will talk about these special characters, which are often called 'metacharacters'.

The first one is the period '**.**'. On the left side of a substitute command, or in a search with '/.../', '**.**' stands for *any* single character. Thus the search

    /x.y/

finds any line where 'x' and 'y' occur separated by a single character, as in

    x+y
    x−y
    x□y
    x.y

and so on. (We will use □ to stand for a space whenever we need to make it visible.)

Since '**.**' matches a single character, that gives you a way to deal with funny characters printed by **l**. Suppose you have a line that, when printed with the **l** command, appears as

    ....   th\07is   ....

and you want to get rid of the \07 (which represents the bell character, by the way).

The most obvious solution is to try

    s/\07//

but this will fail. (Try it.) The brute force solution, which most people would now take, is to re-type the entire line. This is guaranteed, and is actually quite a reasonable tactic if the line in question isn't too big, but for a very long line, re-typing is a bore. This is where the metacharacter '**.**' comes in handy. Since '\07' really represents a single character, if we say

    s/th.is/this/

the job is done. The '**.**' matches the mysterious character between the 'h' and the 'i', *whatever it is.*

Bear in mind that since '**.**' matches any single character, the command

    s/./,/

converts the first character on a line into a ',', which very often is not what you intended.

As is true of many characters in **ed**, the '**.**' has several meanings, depending on its context. This line shows all three:

    .s/./././

The first '**.**' is a line number, the number of the line we are editing, which is called 'line dot'. (We will discuss line dot more in Section 3.) The second '**.**' is a metacharacter that matches any single character on that line. The third '**.**' is the only one that really is an honest literal period. On the *right* side of a substitution, '**.**' is not special. If you apply this command to the line

    Now is the time.

the result will be

    .ow is the time.

which is probably not what you intended.

The Backslash '\'

Since a period means 'any character', the question naturally arises of what to do when you really want a period. For example, how do you convert the line

    Now is the time.

into

    Now is the time?

The backslash '\' does the job. A backslash turns off any special meaning that the next character might have; in particular, '**\.**' converts the '**.**' from a 'match anything' into a period, so you can use it to replace the period in

    Now is the time.

like this:

    s/\./?/

The pair of characters '**\.**' is considered by **ed** to be a single real period.

The backslash can also be used when searching for lines that contain a special character. Suppose you are looking for a line that contains

    .PP

The search

    /.PP/

isn't adequate, for it will find a line like

    THE APPLICATION OF ...

because the '**.**' matches the letter 'A'. But if you say

    /\.PP/

you will find only lines that contain '**.**PP'.

The backslash can also be used to turn off special meanings for characters other than '**.**'. For example, consider finding a line that contains a backslash. The search

    /\/

won't work, because the '\' isn't a literal '\', but instead means that the second '/' no longer delimits the search. But by preceding a backslash with another one, you can search for a literal backslash. Thus

    /\\/

does work. Similarly, you can search for a forward slash '/' with

    /\//

The backslash turns off the meaning of the immediately following '/' so that it doesn't terminate the /.../ construction prematurely.

As an exercise, before reading further, find two substitute commands each of which will convert the line

    \x\.\y

into the line

    \x\y

Here are several solutions; verify that each works as advertised.

    s/\\\.//
    s/x../x/
    s/..y/y/

A couple of miscellaneous notes about backslashes and special characters. First, you can use any character to delimit the pieces of an **s** command: there is nothing sacred about slashes. (But you must use slashes for context searching.) For instance, in a line that contains a lot of slashes already, like

    //exec //sys.fort.go // etc...

you could use a colon as the delimiter — to delete all the slashes, type

    s:/::g

Second, if # and @ are your character erase and line kill characters, you have to type \# and \@; this is true whether you're talking to **ed** or any other program.

When you are adding text with **a** or **i** or **c**, backslash is not special, and you should only put in one backslash for each one you really want.

The Dollar Sign '$'

The next metacharacter, the '$', stands for 'the end of the line'. As its most obvious use, suppose you have the line

    Now is the

and you wish to add the word 'time' to the end. Use the $ like this:

    s/$/□time/

to get

    Now is the time

Notice that a space is needed before 'time' in the substitute command, or you will get

    Now is thetime

As another example, replace the second comma in the following line with a period without altering the first:

    Now is the time, for all good men,

The command needed is

s/,$/./

The $ sign here provides context to make specific which comma we mean. Without it, of course, the **s** command would operate on the first comma to produce

Now is the time. for all good men,

As another example, to convert

Now is the time.

into

Now is the time?

as we did earlier, we can use

s/.$/?/

Like '.', the '$' has multiple meanings depending on context. In the line

$s/$/$/

the first '$' refers to the last line of the file, the second refers to the end of that line, and the third is a literal dollar sign, to be added to that line.

The Circumflex '^'

The circumflex (or hat or caret) '^' stands for the beginning of the line. For example, suppose you are looking for a line that begins with 'the'. If you simply say

/the/

you will in all likelihood find several lines that contain 'the' in the middle before arriving at the one you want. But with

/^the/

you narrow the context, and thus arrive at the desired one more easily.

The other use of '^' is of course to enable you to insert something at the beginning of a line:

s/^/□/

places a space at the beginning of the current line.

Metacharacters can be combined. To search for a line that contains *only* the characters

.PP

you can use the command

/^\.PP$/

The Star '*'

Suppose you have a line that looks like this:

text x   y *text*

where *text* stands for lots of text, and there are some indeterminate number of spaces between the **x** and the **y**. Suppose the job is to replace all the spaces between **x** and **y** by a single space. The line is too long to retype, and there are too many spaces to count. What now?

This is where the metacharacter '*' comes in handy. A character followed by a star stands for as many consecutive occurrences of that character as possible. To refer to all the spaces at once, say

s/x□*y/x□y/

The construction '□*' means 'as many spaces as possible'. Thus 'x□*y' means 'an x, as many spaces as possible, then a y'.

The star can be used with any character, not just space. If the original example was instead

text x−−−−−−−−y *text*

then all '−' signs can be replaced by a single space with the command

s/x−*y/x□y/

Finally, suppose that the line was

text x..................y *text*

Can you see what trap lies in wait for the unwary? If you blindly type

s/x.*y/x□y/

what will happen? The answer, naturally, is that it depends. If there are no other x's or y's on the line, then everything works, but it's blind luck, not good management. Remember that '.' matches *any* single character? Then '.*' matches as many single characters as possible, and unless you're careful, it can eat up a lot more of the line than you expected. If the line was, for example, like this:

text x *text* x................y *text* y *text*

then saying

s/x.*y/x□y/

will take everything from the *first* 'x' to the *last* 'y', which, in this example, is undoubtedly more than you wanted.

The solution, of course, is to turn off the special meaning of '.' with '\.':

s/x\.*y/x□y/

Now everything works, for '\.*' means 'as many *periods* as possible'.

There are times when the pattern '.*' is exactly what you want. For example, to change

Now is the time for all good men ....

into

Now is the time.

use '.*' to eat up everything after the 'for':

s/□for.*/./

There are a couple of additional pitfalls associated with '*' that you should be aware of. Most notable is the fact that 'as many as possible' means *zero* or more. The fact that zero is a legitimate possibility is sometimes rather surprising. For example, if our line contained

text xy *text* x y *text*

and we said

s/x□*y/x□y/

the *first* 'xy' matches this pattern, for it consists of an 'x', zero spaces, and a 'y'. The result is that the substitute acts on the first 'xy', and does not touch the later one that actually contains some intervening spaces.

The way around this, if it matters, is to specify a pattern like

/x□□*y/

which says 'an x, a space, then as many more spaces as possible, then a y', in other words, one or more spaces.

The other startling behavior of '*' is again related to the fact that zero is a legitimate number of occurrences of something followed by a star. The command

s/x*/y/g

when applied to the line

abcdef

produces

yaybycydyeyfy

which is almost certainly not what was intended. The reason for this behavior is that zero is a legal number of matches, and there are no x's at the beginning of the line (so that gets converted into a 'y'), nor between the 'a' and the 'b' (so that gets converted into a 'y'), nor ... and so on. Make sure you really want zero matches; if not, in this case write

s/xx*/y/g

'xx*' is one or more x's.

The Brackets '[ ]'

Suppose that you want to delete any numbers that appear at the beginning of all lines of a file. You might first think of trying a series of commands like

```
1,$s/^1*//
1,$s/^2*//
1,$s/^3*//
```

and so on, but this is clearly going to take forever if the numbers are at all long. Unless you want to repeat the commands over and over until finally all numbers are gone, you must get all the digits on one pass. This is the purpose of the brackets [and].

The construction

[0123456789]

matches any single digit — the whole thing is called a 'character class'. With a character class, the job is easy. The pattern '[0123456789]*' matches zero or more digits (an entire number), so

1,$s/^[0123456789]*//

deletes all digits from the beginning of all lines.

Any characters can appear within a character class, and just to confuse the issue there are essentially no special characters inside the brackets; even the backslash doesn't have a special meaning. To search for special characters, for example, you can say

/[.\$^[]/

Within [...], the '[' is not special. To get a ']' into a character class, make it the first character.

It's a nuisance to have to spell out the digits, so you can abbreviate them as [0−9]; similarly, [a−z] stands for the lower case letters, and [A−Z] for upper case.

As a final frill on character classes, you can specify a class that means 'none of the following characters'. This is done by beginning the class with a '^':

[^0−9]

stands for 'any character *except* a digit'. Thus you might find the first line that doesn't begin with a tab or space by a search like

/^[^(space)(tab)]/

Within a character class, the circumflex has a special meaning only if it occurs at the beginning. Just to convince yourself, verify that

/^[^^]/

finds a line that doesn't begin with a circumflex.

The Ampersand '&'

The ampersand '&' is used primarily to save typing. Suppose you have the line

Now is the time

and you want to make it

    Now is the best time

Of course you can always say

    s/the/the best/

but it seems silly to have to repeat the 'the'. The '&' is used to eliminate the repetition. On the *right* side of a substitute, the ampersand means 'whatever was just matched', so you can say

    s/the/& best/

and the '&' will stand for 'the'. Of course this isn't much of a saving if the thing matched is just 'the', but if it is something truly long or awful, or if it is something like '.*' which matches a lot of text, you can save some tedious typing. There is also much less chance of making a typing error in the replacement text. For example, to parenthesize a line, regardless of its length,

    s/.*/(&)/

The ampersand can occur more than once on the right side:

    s/the/& best and & worst/

makes

    Now is the best and the worst time

and

    s/.*/&? &!!/

converts the original line into

    Now is the time? Now is the time!!

To get a literal ampersand, naturally the backslash is used to turn off the special meaning:

    s/ampersand/\&/

converts the word into the symbol. Notice that '&' is not special on the left side of a substitute, only on the *right* side.

Substituting Newlines

ed provides a facility for splitting a single line into two or more shorter lines by 'substituting in a newline'. As the simplest example, suppose a line has gotten unmanageably long because of editing (or merely because it was unwisely typed). If it looks like

    text xy *text*

you can break it between the 'x' and the 'y' like this:

    s/xy/x\
    y/

This is actually a single command, although it is typed on two lines. Bearing in mind that '\' turns off special meanings, it seems relatively intuitive that a '\' at the end of a line would make the newline there no longer special.

You can in fact make a single line into several lines with this same mechanism. As a large example, consider underlining the word 'very' in a long line by splitting 'very' onto a separate line, and preceding it by the **roff** or **nroff** formatting command '.ul'.

    text a very big *text*

The command

    s/□very□/\
    .ul\
    very\
    /

converts the line into four shorter lines, preceding the word 'very' by the line '.ul', and eliminating the spaces around the 'very', all at the same time.

When a newline is substituted in, dot is left pointing at the last line created.

Joining Lines

Lines may also be joined together, but this is done with the **j** command instead of **s**. Given the lines

    Now is
    □the time

and supposing that dot is set to the first of them, then the command

    j

joins them together. No blanks are added, which is why we carefully showed a blank at the beginning of the second line.

All by itself, a **j** command joins line dot to line dot+1, but any contiguous set of lines can be joined. Just specify the starting and ending line numbers. For example,

    1,$jp

joins all the lines into one big one and prints it. (More on line numbers in Section 3.)

Rearranging a Line with \(... \)

(This section should be skipped on first reading.) Recall that '&' is a shorthand that stands for whatever was matched by the left side of an **s** command. In much the same way you can capture separate pieces of what was matched; the only difference is that you have to specify on the left side just what pieces you're interested in.

Suppose, for instance, that you have a file of lines that consist of names in the form

    Smith, A. B.
    Jones, C.

and so on, and you want the initials to precede the name, as in

    A. B. Smith
    C. Jones

It is possible to do this with a series of editing commands, but it is tedious and error-prone. (It is instructive to figure out how it is done, though.)

The alternative is to 'tag' the pieces of the pattern (in this case, the last name, and the initials), and then rearrange the pieces. On the left side of a substitution, if part of the pattern is enclosed between \(and \), whatever matched that part is remembered, and available for use on the right side. On the right side, the symbol '\1' refers to whatever matched the first \(...\) pair, '\2' to the second \(...\), and so on.

The command

```
1,$s/^\([^,]*\),␣*\(.*\)/\2␣\1/
```

although hard to read, does the job. The first \(...\) matches the last name, which is any string up to the comma; this is referred to on the right side with '\1'. The second \(...\) is whatever follows the comma and any spaces, and is referred to as '\2'.

Of course, with any editing sequence this complicated, it's foolhardy to simply run it and hope. The global commands **g** and **v** discussed in section 4 provide a way for you to print exactly those lines which were affected by the substitute command, and thus verify that it did what you wanted in all cases.

3. LINE ADDRESSING IN THE EDITOR

The next general area we will discuss is that of line addressing in **ed**, that is, how you specify what lines are to be affected by editing commands. We have already used constructions like

```
1,$s/x/y/
```

to specify a change on all lines. And most users are long since familiar with using a single newline (or return) to print the next line, and with

```
/thing/
```

to find a line that contains 'thing'. Less familiar, surprisingly enough, is the use of

```
?thing?
```

to scan *backwards* for the previous occurrence of 'thing'. This is especially handy when you realize that the thing you want to operate on is back up the page from where you are currently editing.

The slash and question mark are the only characters you can use to delimit a context search, though you can use essentially any character in a substitute command.

Address Arithmetic

The next step is to combine the line numbers like '.', '$', '/.../' and '?...?' with '+' and '–'. Thus

```
$-1
```

is a command to print the next to last line of the current file (that is, one line before line '$'). For example, to recall how far you got in a previous editing session,

```
$-5,$p
```

prints the last six lines. (Be sure you understand why it's six, not five.) If there aren't six, of course, you'll get an error message.

As another example,

```
.-3,.+3p
```

prints from three lines before where you are now (at line dot) to three lines after, thus giving you a bit of context. By the way, the '+' can be omitted:

```
.-3,.3p
```

is absolutely identical in meaning.

Another area in which you can save typing effort in specifying lines is to use '–' and '+' as line numbers by themselves.

```
-
```

by itself is a command to move back up one line in the file. In fact, you can string several minus signs together to move back up that many lines:

```
---
```

moves up three lines, as does '–3'. Thus

```
-3,+3p
```

is also identical to the examples above.

Since '–' is shorter than '.–1', constructions like

```
-,.s/bad/good/
```

are useful. This changes 'bad' to 'good' on the previous line and on the current line.

'+' and '–' can be used in combination with searches using '/.../' and '?...?', and with '$'. The search

```
/thing/--
```

finds the line containing 'thing', and positions you two lines before it.

Repeated Searches

Suppose you ask for the search

```
/horrible thing/
```

and when the line is printed you discover that it isn't the horrible thing that you wanted, so it is necessary to repeat the search again. You don't have to re-type the search, for the construction

```
//
```

is a shorthand for 'the previous thing that was searched for', whatever it was. This can be repeated as many times as necessary. You can also go backwards:

```
??
```

searches for the same thing, but in the reverse direction.

Not only can you repeat the search, but you can use '//' as the left side of a substitute command, to mean 'the most recent pattern'.

```
/horrible thing/
 .... ed prints line with 'horrible thing' ...
s//good/p
```

To go backwards and change a line, say

```
??s//good/
```

Of course, you can still use the '&' on the right hand side of a substitute to stand for whatever got matched:

```
//s//& &/p
```

finds the next occurrence of whatever you searched for last, replaces it by two copies of itself, then prints the line just to verify that it worked.

Default Line Numbers and the Value of Dot

One of the most effective ways to speed up your editing is always to know what lines will be affected by a command if you don't specify the lines it is to act on, and on what line you will be positioned (i.e., the value of dot) when a command finishes. If you can edit without specifying unnecessary line numbers, you can save a lot of typing.

As the most obvious example, if you issue a search command like

```
/thing/
```

you are left pointing at the next line that contains 'thing'. Then no address is required with commands like **s** to make a substitution on that line, or **p** to print it, or **l** to list it, or **d** to delete it, or **a** to append text after it, or **c** to change it, or **i** to insert text before it.

What happens if there was no 'thing'? Then you are left right where you were — dot is unchanged. This is also true if you were sitting on the only 'thing' when you issued the command. The same rules hold for searches that use '?...?'; the only difference is the direction in which you search.

The delete command **d** leaves dot pointing at the line that followed the last deleted line. When line '$' gets deleted, however, dot points at the *new* line '$'.

The line-changing commands **a**, **c** and **i** by default all affect the current line — if you give no line number with them, **a** appends text after the current line, **c** changes the current line, and **i** inserts text before the current line.

a, **c**, and **i** behave identically in one respect — when you stop appending, changing or inserting, dot points at the last line entered. This is exactly what you want for typing and editing on the fly. For example, you can say

```
a
 ... text ...
 ... botch ...                  (minor error)
.
s/botch/correct/                (fix botched line)
a
 ... more text ...
```

without specifying any line number for the substitute command or for the second append command. Or you can say

```
a
 ... text ...
 ... horrible botch ...         (major error)
.
c                               (replace entire line)
 ... fixed up line ...
```

You should experiment to determine what happens if you add *no* lines with **a**, **c** or **i**.

The **r** command will read a file into the text being edited, either at the end if you give no address, or after the specified line if you do. In either case, dot points at the last line read in. Remember that you can even say **0r** to read a file in at the beginning of the text. (You can also say **0a** or **1i** to start adding text at the beginning.)

The **w** command writes out the entire file. If you precede the command by one line number, that line is written, while if you precede it by two line numbers, that range of lines is written. The **w** command does *not* change dot: the current line remains the same, regardless of what lines are written. This is true even if you say something like

```
/^\.AB/,/^\.AE/w abstract
```

which involves a context search.

Since the **w** command is so easy to use, you should save what you are editing regularly as you go along just in case the system crashes, or in case you do something foolish, like clobbering what you're editing.

The least intuitive behavior, in a sense, is that of the **s** command. The rule is simple — you are left sitting on the last line that got changed. If there were no changes, then dot is unchanged.

To illustrate, suppose that there are three lines in the buffer, and you are sitting on the middle one:

```
x1
x2
x3
```

Then the command

```
–,+s/x/y/p
```

prints the third line, which is the last one changed. But if the three lines had been

```
x1
y2
y3
```

and the same command had been issued while dot pointed at the second line, then the result would be to change and print only the first line, and that is where dot would be set.

Semicolon ';'

Searches with '/.../' and '?...?' start at the current line and move forward or backward respectively until they either find the pattern or get back to the current line. Sometimes this is not what is wanted. Suppose, for example, that the buffer contains lines like this:

```
.
.
.
ab
.
.
.
bc
.
.
```

Starting at line 1, one would expect that the command

```
/a/,/b/p
```

prints all the lines from the 'ab' to the 'bc' inclusive. Actually this is not what happens. *Both* searches (for 'a' and for 'b') start from the same point, and thus they both find the line that contains 'ab'. The result is to print a single line. Worse, if there had been a line with a 'b' in it before the 'ab' line, then the print command would be in error, since the second line number would be less than the first, and it is illegal to try to print lines in reverse order.

This is because the comma separator for line numbers doesn't set dot as each address is processed; each search starts from the same place. In **ed**, the semicolon ';' can be used just like comma, with the single difference that use of a semicolon forces dot to be set at that point as the line numbers are being evaluated. In effect, the semicolon 'moves' dot. Thus in our example above, the command

```
/a/;/b/p
```

prints the range of lines from 'ab' to 'bc', because after the 'a' is found, dot is set to that line, and then 'b' is searched for, starting beyond that line.

This property is most often useful in a very simple situation. Suppose you want to find the *second* occurrence of 'thing'. You could say

```
/thing/
//
```

but this prints the first occurrence as well as the second, and is a nuisance when you know very well that it is only the second one you're interested in. The solution is to say

```
/thing/;//
```

This says to find the first occurrence of 'thing', set dot to that line, then find the second and print only that.

Closely related is searching for the second previous occurrence of something, as in

```
?something?;??
```

Printing the third or fourth or ... in either direction is left as an exercise.

Finally, bear in mind that if you want to find the first occurrence of something in a file, starting at an arbitrary place within the file, it is not sufficient to say

```
1;/thing/
```

because this fails if 'thing' occurs on line 1. But it is possible to say

```
0;/thing/
```

(one of the few places where 0 is a legal line number), for this starts the search at line 1.

Interrupting the Editor

As a final note on what dot gets set to, you should be aware that if you hit the interrupt or delete or rubout or break key while **ed** is doing a command, things are put back together again and your state is restored as much as possible to what it was before the command began. Naturally, some changes are irrevocable — if you are reading or writing a file or making substitutions or deleting lines, these will be stopped in some clean but unpredictable state in the middle (which is why it is not usually wise to stop them). Dot may or may not be changed.

Printing is more clear cut. Dot is not changed until the printing is done. Thus if you print until you see an interesting line, then hit delete, you are *not* sitting on that line or even near it. Dot is left where it was when the **p** command was started.

4. GLOBAL COMMANDS

The global commands **g** and **v** are used to perform one or more editing commands on all lines that either contain (**g**) or don't contain (**v**) a specified pattern.

As the simplest example, the command

```
g/UNIX/p
```

prints all lines that contain the word 'UNIX'. The pattern that goes between the slashes can be anything that could be used in a line search or in a substitute command; exactly the same rules and limitations apply.

As another example, then,

```
g/^\./p
```

prints all the formatting commands in a file (lines that begin with '**.**').

The **v** command is identical to **g**, except that it operates on those line that do *not* contain an occurrence of the pattern. (Don't look too hard for mnemonic significance to the letter 'v'.) So

```
v/^\./p
```

prints all the lines that don't begin with '**.**' — the actual text lines.

The command that follows **g** or **v** can be anything:

```
g/^\./d
```

deletes all lines that begin with '**.**', and

```
g/^$/d
```

deletes all empty lines.

Probably the most useful command that can follow a global is the substitute command, for this can be used to make a change and print each affected line for verification. For example, we could change the word 'Unix' to 'UNIX' everywhere, and verify that it really worked, with

```
g/Unix/s//UNIX/gp
```

Notice that we used '//' in the substitute command to mean 'the previous pattern', in this case, 'Unix'. The **p** command is done on every line that matches the pattern, not just those on which a substitution took place.

The global command operates by making two passes over the file. On the first pass, all lines that match the pattern are marked. On the second pass, each marked line in turn is examined, dot is set to that line, and the command executed. This means that it is possible for the command that follows a **g** or **v** to use addresses, set dot, and so on, quite freely.

```
g/^\.PP/+
```

prints the line that follows each '.PP' command (the signal for a new paragraph in some formatting packages). Remember that '+' means 'one line past dot'. And

```
g/topic/?^\.SH?1
```

searches for each line that contains 'topic', scans backwards until it finds a line that begins '.SH' (a section heading) and prints the line that follows that, thus showing the section headings under which 'topic' is mentioned. Finally,

```
g/^\.EQ/+,/^\.EN/−p
```

prints all the lines that lie between lines beginning with '.EQ' and '.EN' formatting commands.

The **g** and **v** commands can also be preceded by line numbers, in which case the lines searched are only those in the range specified.

Multi-line Global Commands

It is possible to do more than one command under the control of a global command, although the syntax for expressing the operation is not especially natural or pleasant. As an example, suppose the task is to change 'x' to 'y' and 'a' to 'b' on all lines that contain 'thing'. Then

```
g/thing/s/x/y/\
s/a/b/
```

is sufficient. The '\' signals the **g** command that the set of commands continues on the next line; it terminates on the first line that does not end with '\'. (As a minor blemish, you can't use a substitute command to insert a newline within a **g** command.)

You should watch out for this problem: the command

```
g/x/s//y/\
s/a/b/
```

does *not* work as you expect. The remembered pattern is the last pattern that was actually executed, so sometimes it will be 'x' (as expected), and sometimes it will be 'a' (not expected). You must spell it out, like this:

```
g/x/s/x/y/\
s/a/b/
```

It is also possible to execute **a**, **c** and **i** commands under a global command; as with other multi-line constructions, all that is needed is to add a '\' at the end of each line except the last. Thus to add a '.nf' and '.sp' command before each '.EQ' line, type

```
g/^\.EQ/i\
.nf\
.sp
```

There is no need for a final line containing a '**.**' to terminate the **i** command, unless there are further commands being done under the global. On the other hand, it does no harm to put it in either.

5. CUT AND PASTE WITH UNIX COMMANDS

One editing area in which non-programmers seem not very confident is in what might be called 'cut and paste' operations — changing the name of a file, making a copy of a file somewhere else, moving a few lines from one place to another in a file, inserting one file in the middle of another, splitting a file into pieces, and splicing two or more files together.

Yet most of these operations are actually quite easy, if you keep your wits about you and go cautiously. The next several sections talk about cut and paste. We will begin with the UNIX commands for moving entire files around, then discuss **ed** commands for operating on pieces of files.

Changing the Name of a File

You have a file named 'memo' and you want it to be called 'paper' instead. How is it done?

The UNIX program that renames files is called **mv** (for 'move'); it 'moves' the file from one name to another, like this:

```
mv  memo  paper
```

That's all there is to it: **mv** from the old name to the new name.

```
mv  oldname  newname
```

Warning: if there is already a file around with the new name, its present contents will be silently clobbered by the information from the other file. The one exception is that you can't move a file to itself —

```
mv  x  x
```

is illegal.

Making a Copy of a File

Sometimes what you want is a copy of a file — an entirely fresh version. This might be because you want to work on a file, and yet save a copy in case something gets fouled up, or just because you're paranoid.

In any case, the way to do it is with the **cp** command. (**cp** stands for 'copy'; the system is big on short command names, which are appreciated by heavy users, but sometimes a strain for novices.) Suppose you have a file called 'good' and you want to save a copy before you make some dramatic editing changes. Choose a name — 'savegood' might be acceptable — then type

```
cp  good  savegood
```

This copies 'good' onto 'savegood', and you now have two identical copies of the file 'good'. (If 'savegood' previously contained something, it gets overwritten.)

Now if you decide at some time that you want to get back to the original state of 'good', you can say

```
mv  savegood  good
```

(if you're not interested in 'savegood' any more), or

```
cp  savegood  good
```

if you still want to retain a safe copy.

In summary, **mv** just renames a file; **cp** makes a duplicate copy. Both of them clobber the 'target' file if it already exists, so you had better be sure that's what you want to do *before* you do it.

Removing a File

If you decide you are really done with a file forever, you can remove it with the **rm** command:

```
rm  savegood
```

throws away (irrevocably) the file called 'savegood'.

Putting Two or More Files Together

The next step is the familiar one of collecting two or more files into one big one. This will be needed, for example, when the author of a paper decides that several sections need to be combined into one. There are several ways to do it, of which the cleanest, once you get used to it, is a program called **cat**. (Not *all* programs have two-letter names.) **cat** is short for 'concatenate', which is exactly what we want to do.

Suppose the job is to combine the files 'file1' and 'file2' into a single file called 'bigfile'. If you say

```
cat  file
```

the contents of 'file' will get printed on your terminal. If you say

```
cat  file1  file2
```

the contents of 'file1' and then the contents of 'file2' will *both* be printed on your terminal, in that order. So **cat** combines the files, all right, but it's not much help to print them on the terminal — we want them in 'bigfile'.

Fortunately, there is a way. You can tell the system that instead of printing on your terminal, you want the same information put in a file. The way to do it is to add to the command line the character > and the name of the file where you want the output to go. Then you can say

```
cat  file1  file2  >bigfile
```

and the job is done. (As with **cp** and **mv**, you're putting something into 'bigfile', and anything that was already there is destroyed.)

This ability to 'capture' the output of a program is one of the most useful aspects of the system. Fortunately it's not limited to the **cat** program — you can use it with *any* program that prints on your terminal. We'll see some more uses for it in a moment.

Naturally, you can combine several files, not just two:

```
cat  file1  file2  file3  ...  >bigfile
```

collects a whole bunch.

Question: is there any difference between

```
cp  good  savegood
```

and

```
cat  good  >savegood
```

Answer: for most purposes, no. You might reasonably ask why there are two programs in that case, since **cat** is obviously all you need. The answer is that **cp** will do some other things as well, which you can investigate for yourself by reading the manual. For now we'll stick to simple usages.

Adding Something to the End of a File

Sometimes you want to add one file to the end of another. We have enough building blocks now that you can do it; in fact before reading further it would be valuable if you figured out how. To be specific, how would you use **cp**, **mv** and/or **cat** to add the file 'good1' to the end of the file 'good'?

You could try

```
cat  good  good1  >temp
mv  temp  good
```

which is probably most direct. You should also understand why

```
cat  good  good1  >good
```

doesn't work. (Don't practice with a good 'good'!)

The easy way is to use a variant of >, called >>. In fact, >> is identical to > except that instead of clobbering the old file, it simply tacks stuff on at the end. Thus you could say

```
cat  good1  >>good
```

and 'good1' is added to the end of 'good'. (And if 'good' didn't exist, this makes a copy of 'good1' called 'good'.)

6. CUT AND PASTE WITH THE EDITOR

Now we move on to manipulating pieces of files — individual lines or groups of lines. This is another area where new users seem unsure of themselves.

Filenames

The first step is to ensure that you know the **ed** commands for reading and writing files. Of course you can't go very far without knowing **r** and **w**. Equally useful, but less well known, is the 'edit' command **e**. Within **ed**, the command

```
e  newfile
```

says 'I want to edit a new file called *newfile,* without leaving the editor.' The **e** command discards whatever you're currently working on and starts over on *newfile.* It's exactly the same as if you had quit with the **q** command, then re-entered **ed** with a new file name, except that if you have a pattern remembered, then a command like **//** will still work.

If you enter **ed** with the command

```
ed  file
```

ed remembers the name of the file, and any subsequent **e**, **r** or **w** commands that don't contain a filename will refer to this remembered file. Thus

```
ed  file1
 ... (editing) ...
w          (writes back in file1)
e  file2   (edit new file, without leaving editor)
 ... (editing on file2) ...
w          (writes back on file2)
```

(and so on) does a series of edits on various files without ever leaving **ed** and without typing the name of any file more than once. (As an aside, if you examine the sequence of commands here, you can see why many UNIX systems use **e** as a synonym for **ed**.)

You can find out the remembered file name at any time with the **f** command; just type **f** without a file name. You can also change the name of the remembered file name with **f**; a useful sequence is

```
ed  precious
f  junk
 ... (editing) ...
```

which gets a copy of a precious file, then uses **f** to guarantee that a careless **w** command won't clobber the original.

Inserting One File into Another

Suppose you have a file called 'memo', and you want the file called 'table' to be inserted just after the reference to Table 1. That is, in 'memo' somewhere is a line that says

```
Table 1 shows that ...
```

and the data contained in 'table' has to go there, probably so it will be formatted properly by **nroff** or **troff**. Now what?

This one is easy. Edit 'memo', find 'Table 1', and add the file 'table' right there:

```
ed  memo
/Table 1/
Table 1 shows that ... [response from ed]
.r  table
```

The critical line is the last one. As we said earlier, the **r** command reads a file; here you asked for it to be read in right after line dot. An **r** command without any address adds lines at the end, so it is the same as **$r**.

Writing out Part of a File

The other side of the coin is writing out part of the document you're editing. For example, maybe you want to split out into a separate file that table from the previous example, so it can be formatted and tested separately. Suppose that in the file being edited we have

```
.TS
 ...[lots of stuff]
.TE
```

which is the way a table is set up for the **tbl** program. To isolate the table in a separate file called 'table', first find the start of the table (the '.TS' line), then write out the interesting part:

```
/^\.TS/
.TS  [ed prints the line it found]
.,/^\.TE/w table
```

and the job is done. If you are confident, you can do it all at once with

```
/^\.TS/;/^\.TE/w table
```

The point is that the **w** command can write out a group of lines, instead of the whole file. In fact, you can write out a single line if you like; just give one line number instead of two. For example, if you have just typed a horribly complicated line and you know that it (or something like it) is going to be needed later, then save it — don't re-type it. In the editor, say

```
a
...lots of stuff...
...horrible line...
.
.w  temp
a
...more stuff...
.
.r temp
a
...more stuff...
.
```

This last example is worth studying, to be sure you appreciate what's going on.

Moving Lines Around

Suppose you want to move a paragraph from its present position in a paper to the end. How would you do it? As a concrete example, suppose each paragraph in the paper begins with the formatting command '.PP'. Think about it and write down the details before reading on.

The brute force way (not necessarily bad) is to write the paragraph onto a temporary file, delete it from its current position, then read in the temporary file at the end. Assuming that you are sitting on the '.PP' command that begins the paragraph, this is the sequence of commands:

```
.,/^\.PP/−w temp
.,//−d
$r temp
```

That is, from where you are now ('**.**') until one line before the next '**.**PP' ('/^\.PP/−') write onto 'temp'. Then delete the same lines. Finally, read 'temp' at the end.

As we said, that's the brute force way. The easier way (often) is to use the *move* command **m** that **ed** provides — it lets you do the whole set of operations at one crack, without any temporary file.

The **m** command is like many other **ed** commands in that it takes up to two line numbers in front that tell what lines are to be affected. It is also *followed* by a line number that tells where the lines are to go. Thus

```
line1, line2 m line3
```

says to move all the lines between 'line1' and 'line2' after 'line3'. Naturally, any of 'line1' etc., can be patterns between slashes, $ signs, or other ways to specify lines.

Suppose again that you're sitting at the first line of the paragraph. Then you can say

```
.,/^\.PP/−m$
```

That's all.

As another example of a frequent operation, you can reverse the order of two adjacent lines by moving the first one to after the second. Suppose that you are positioned at the first. Then

```
m+
```

does it. It says to move line dot to after one line after line dot. If you are positioned on the second line,

```
m−−
```

does the interchange.

As you can see, the **m** command is more succinct and direct than writing, deleting and re-reading. When is brute force better anyway? This is a matter of personal taste — do what you have most confidence in. The main difficulty with the **m** command is that if you use patterns to specify both the lines you are moving and the target, you have to take care that you specify them properly, or you may well not move the lines you thought you did. The result of a botched **m** command can be a ghastly mess. Doing the job a step at a time makes it easier for you to verify at each step that you accomplished what you wanted to. It's also a good idea to issue a **w** command before doing anything complicated; then if you goof, it's easy to back up to where you were.

Marks

ed provides a facility for marking a line with a particular name so you can later reference it by name regardless of its actual line number. This can be handy for moving lines, and for keeping track of them as they move. The *mark* command is **k**; the command

```
kx
```

marks the current line with the name 'x'. If a line number precedes the **k**, that line is marked. (The mark name must be a single lower case letter.) Now you can refer to the marked line with the address

```
′x
```

Marks are most useful for moving things around. Find the first line of the block to be moved, and mark it with *′a.* Then find the last line and mark it with *′b.* Now position yourself at the place where the stuff is to go and say

```
′a,′bm.
```

Bear in mind that only one line can have a particular mark name associated with it at any given time.

Copying Lines

We mentioned earlier the idea of saving a line that was hard to type or used often, so as to cut down on typing time. Of course this could be more than one line; then the saving is presumably even greater.

ed provides another command, called **t** (for 'transfer') for making a copy of a group of one or more lines at any point. This is often easier than writing and reading.

The **t** command is identical to the **m** command, except that instead of moving lines it simply duplicates them at the place you named. Thus

```
1,$t$
```

duplicates the entire contents that you are editing. A more common use for **t** is for creating a series of lines that differ only slightly. For example, you can say

```
a
.......... x ......... (long line)
.
t.            (make a copy)
s/x/y/        (change it a bit)
t.            (make third copy)
s/y/z/        (change it a bit)
```

and so on.

The Temporary Escape '!'

Sometimes it is convenient to be able to temporarily escape from the editor to do some other UNIX command, perhaps one of the file copy or move commands discussed in section 5, without leaving the editor. The 'escape' command **!** provides a way to do this.

If you say

```
!any UNIX command
```

your current editing state is suspended, and the UNIX command you asked for is executed. When the command finishes, **ed** will signal you by printing another **!**; at that point you can resume editing.

You can really do *any* UNIX command, including another **ed**. (This is quite common, in fact.) In this case, you can even do another **!**.

7. SUPPORTING TOOLS

There are several tools and techniques that go along with the editor, all of which are relatively easy once you know how **ed** works, because they are all based on the editor. In this section we will give some fairly cursory examples of these tools, more to indicate their existence than to provide a complete tutorial. More information on each can be found in [3].

Grep

Sometimes you want to find all occurrences of some word or pattern in a set of files, to edit them or perhaps just to verify their presence or absence. It may be possible to edit each file separately and look for the pattern of interest, but if there are many files this can get very tedious, and if the files are really big, it may be impossible because of limits in **ed**.

The program **grep** was invented to get around these limitations. The search patterns that we have described in the paper are often called 'regular expressions', and 'grep' stands for

```
g/re/p
```

That describes exactly what **grep** does — it prints every line in a set of files that contains a particular pattern. Thus

```
grep ′thing′ file1 file2 file3 ...
```

finds 'thing' wherever it occurs in any of the files 'file1', 'file2', etc. **grep** also indicates the file in which the line was found, so you can later edit it if you like.

The pattern represented by 'thing' can be any pattern you can use in the editor, since **grep** and **ed** use exactly the same mechanism for pattern searching. It is wisest always to enclose the pattern in the single quotes ′...′ if it contains any non-alphabetic characters, since many such characters also mean something special to the UNIX command interpreter (the 'shell'). If you don't quote them, the command interpreter will try to interpret them before **grep** gets a chance.

There is also a way to find lines that *don't* contain a pattern:

```
grep −v ′thing′ file1 file2 ...
```

finds all lines that don't contains 'thing'. The **−v** must occur in the position shown. Given **grep** and **grep −v**, it is possible to do things like selecting all lines that contain some combination of patterns. For example, to get all lines that contain 'x' but not 'y':

```
grep x file... | grep −v y
```

(The notation | is a 'pipe', which causes the output of the first command to be used as input to the second command; see [2].)

Editing Scripts

If a fairly complicated set of editing operations is to be done on a whole set of files, the easiest thing to do is to make up a 'script', i.e., a file that contains the operations you want to perform, then apply this script to each file in turn.

For example, suppose you want to change every 'Unix' to 'UNIX' and every 'Gcos' to 'GCOS' in a large number of files. Then put into the file 'script' the lines

```
g/Unix/s//UNIX/g
g/Gcos/s//GCOS/g
w
q
```

Now you can say

```
ed file1 <script
ed file2 <script
...
```

This causes **ed** to take its commands from the prepared script. Notice that the whole job has to be planned in advance.

And of course by using the UNIX command interpreter, you can cycle through a set of files automatically, with varying degrees of ease.

Sed

sed ('stream editor') is a version of the editor with restricted capabilities but which is capable of processing unlimited amounts of input. Basically **sed** copies its input to its output, applying one or more editing commands to each line of input.

As an example, suppose that we want to do the 'Unix' to 'UNIX' part of the example given above, but without rewriting the files. Then the command

```
sed ′s/Unix/UNIX/g′ file1 file2 ...
```

applies the command 's/Unix/UNIX/g' to all lines from 'file1', 'file2', etc., and copies all lines to the output. The advantage of using **sed** in such a case is that it can be used with input too large for **ed** to handle. All the output can be collected in one place, either in a file or perhaps piped into another program.

If the editing transformation is so complicated that more than one editing command is needed, commands can be supplied from a file, or on the command line, with a slightly more complex syntax. To take commands from a file, for example,

```
sed −f cmdfile input−files...
```

sed has further capabilities, including conditional testing and branching, which we cannot go into here.

Acknowledgement

I am grateful to Ted Dolotta for his careful reading and valuable suggestions.

References

[1] Brian W. Kernighan, *A Tutorial Introduction to the UNIX Text Editor,* Bell Laboratories internal memorandum.

[2] Brian W. Kernighan, *UNIX For Beginners,* Bell Laboratories internal memorandum.

[3] Ken L. Thompson and Dennis M. Ritchie, *The UNIX Programmer's Manual.* Bell Laboratories.

An Introduction to the UNIX Shell

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The *shell* is a command programming language that provides an interface to the UNIX† operating system. Its features include control-flow primitives, parameter passing, variables and string substitution. Constructs such as *while, if then else, case* and *for* are available. Two-way communication is possible between the *shell* and commands. String-valued parameters, typically file names or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as shell input.

The *shell* can modify the environment in which commands run. Input and output can be redirected to files, and processes that communicate through 'pipes' can be invoked. Commands are found by searching directories in the file system in a sequence that can be defined by the user. Commands can be read either from the terminal or from a file, which allows command procedures to be stored for later use.

November 12, 1978

†UNIX is a Trademark of Bell Laboratories.

An Introduction to the UNIX Shell

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

1.0 Introduction

The shell is both a command language and a programming language that provides an interface to the UNIX operating system. This memorandum describes, with examples, the UNIX shell. The first section covers most of the everyday requirements of terminal users. Some familiarity with UNIX is an advantage when reading this section; see, for example, "UNIX for beginners".[1] Section 2 describes those features of the shell primarily intended for use within shell procedures. These include the control-flow primitives and string-valued variables provided by the shell. A knowledge of a programming language would be a help when reading this section. The last section describes the more advanced features of the shell. References of the form "see *pipe* (2)" are to a section of the UNIX manual.[2]

1.1 Simple commands

Simple commands consist of one or more words separated by blanks. The first word is the name of the command to be executed; any remaining words are passed as arguments to the command. For example,

```
who
```

is a command that prints the names of users logged in. The command

```
ls –l
```

prints a list of files in the current directory. The argument *–l* tells *ls* to print status information, size and the creation date for each file.

1.2 Background commands

To execute a command the shell normally creates a new *process* and waits for it to finish. A command may be run without waiting for it to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file *pgm.c*. The trailing **&** is an operator that instructs the shell not to wait for the command to finish. To help keep track of such a process the shell reports its process number following its creation. A list of currently active processes may be obtained using the *ps* command.

1.3 Input output redirection

Most commands produce output on the standard output that is initially connected to the terminal. This output may be sent to a file by writing, for example,

```
ls –l >file
```

The notation >*file* is interpreted by the shell and is not passed as an argument to *ls.* If *file* does not exist then the shell creates it; otherwise the original contents of *file* are replaced with the output from *ls.* Output may be appended to a file using the notation

```
ls –l >>file
```

In this case *file* is also created if it does not already exist.

The standard input of a command may be taken from a file instead of the terminal by writing, for example,

    wc <file

The command *wc* reads its standard input (in this case redirected from *file*) and prints the number of characters, words and lines found. If only the number of lines is required then

    wc –l <file

could be used.

1.4 Pipelines and filters

The standard output of one command may be connected to the standard input of another by writing the 'pipe' operator, indicated by |, as in,

    ls –l | wc

Two commands connected in this way constitute a *pipeline* and the overall effect is the same as

    ls –l >file; wc <file

except that no *file* is used. Instead the two processes are connected by a pipe (see *pipe* (2)) and are run in parallel. Pipes are unidirectional and synchronization is achieved by halting *wc* when there is nothing to read and halting *ls* when the pipe is full.

A *filter* is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, *grep,* selects from its input those lines that contain some specified string. For example,

    ls | grep old

prints those lines, if any, of the output from *ls* that contain the string *old.* Another useful filter is *sort*. For example,

    who | sort

will print an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands, for example,

    ls | grep old | wc –l

prints the number of file names in the current directory containing the string *old.*

1.5 File name generation

Many commands accept arguments which are file names. For example,

    ls –l main.c

prints information relating to the file *main.c* .

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

    ls –l ∗.c

generates, as arguments to *ls,* all file names in the current directory that end in *.c* . The character ∗ is a pattern that will match any string including the null string. In general *patterns* are specified as follows.

∗ Matches any string of characters including the null string.

? Matches any single character.

[...] Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

[a–z]*

matches all names in the current directory beginning with one of the letters *a* through *z.*

/usr/fred/test/?

matches all names in the directory **/usr/fred/test** that consist of a single character. If no file name is found that matches the pattern then the pattern is passed, unchanged, as an argument.

This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

echo /usr/fred/*/core

finds and prints the names of all *core* files in sub-directories of **/usr/fred.** (*echo* is a standard UNIX command that prints its arguments, separated by blanks.) This last feature can be expensive, requiring a scan of all sub-directories of **/usr/fred.**

There is one exception to the general rules given for patterns. The character '**.**' at the start of a file name must be explicitly matched.

echo *

will therefore echo all file names in the current directory not beginning with '**.**' .

echo **.***

will echo all those file names that begin with '**.**' . This avoids inadvertent matching of the names '**.**' and '**..**' which mean 'the current directory' and 'the parent directory' respectively. (Notice that *ls* suppresses information for the files '**.**' and '**..**' .)

1.6 Quoting

Characters that have a special meaning to the shell, such as < > *** ? | &,** are called metacharacters. A complete list of metacharacters is given in appendix B. Any character preceded by a \ is *quoted* and loses its special meaning, if any. The \ is elided so that

echo \?

will echo a single **?,** and

echo \\

will echo a single **.** To allow long strings to be continued over more than one line the sequence **\newline** is ignored.

\ is convenient for quoting single characters. When more than one character needs quoting the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

echo xx´****´xx

will echo

xx****xx

The quoted string may not contain a single quote but may contain newlines, which are preserved. This quoting mechanism is the most simple and is recommended for casual use.

A third quoting mechanism using double quotes is also available that prevents interpretation of some but not all metacharacters. Discussion of the details is deferred to section 3.4 .

1.7 Prompting

When the shell is used from a terminal it will issue a prompt before reading a command. By default this prompt is '**$** '. It may be changed by saying, for example,

```
PS1=yesdear
```

that sets the prompt to be the string *yesdear*. If a newline is typed and further input is needed then the shell will issue the prompt '**>** '. Sometimes this can be caused by mistyping a quote mark. If it is unexpected then an interrupt (DEL) will return the shell to read another command. This prompt may be changed by saying, for example,

```
PS2=more
```

1.8 The shell and login

Following *login* (1) the shell is called to read and execute commands typed at the terminal. If the user's login directory contains the file **.profile** then it is assumed to contain commands and is read by the shell before reading any commands from the terminal.

1.9 Summary

- **ls**
  Print the names of files in the current directory.
- **ls >file**
  Put the output from *ls* into *file.*
- **ls | wc –l**
  Print the number of files in the current directory.
- **ls | grep old**
  Print those file names containing the string *old.*
- **ls | grep old | wc –l**
  Print the number of files whose name contains the string *old.*
- **cc pgm.c &**
  Run *cc* in the background.

2.0 Shell procedures

The shell may be used to read and execute commands contained in a file. For example,

```
sh file [ args ... ]
```

calls the shell to read commands from *file.* Such a file is called a *command procedure* or *shell procedure.* Arguments may be supplied with the call and are referred to in *file* using the positional parameters **$1, $2, ...** For example, if the file *wg* contains

```
who | grep $1
```

then

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

UNIX files have three independent attributes, *read, write* and *execute.* The UNIX command *chmod* (1) may be used to make a file executable. For example,

```
chmod +x wg
```

will ensure that the file *wg* has execute status. Following this, the command

```
wg fred
```

is equivalent to

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably. In either case a new process is created to run the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as **$#.** The name of the file being executed is available as **$0.**

A special shell parameter **$*** is used to substitute for all positional parameters except **$0.** A typical use of this is to provide some default arguments, as in,

```
nroff −T450 −ms $*
```

which simply prepends some arguments to those already given.

2.1 Control flow - for

A frequent use of shell procedures is to loop through the arguments (**$1, $2, ...**) executing commands once for each argument. An example of such a procedure is *tel* that searches the file **/usr/lib/telnos** that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of *tel* is

```
for i
do grep $i /usr/lib/telnos; done
```

The command

```
tel fred
```

prints those lines in **/usr/lib/telnos** that contain the string *fred*.

```
tel fred bert
```

prints those lines containing *fred* followed by those for *bert.*

The **for** loop notation is recognized by the shell and has the general form

> **for** *name* **in** *w1 w2* **...**
> **do** *command-list*
> **done**

A *command-list* is a sequence of one or more simple commands separated or terminated by a newline or semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a newline or semicolon. *name* is a shell variable that is set to the words *w1 w2* **...** in turn each time the *command-list* following **do** is executed. If **in** *w1 w2* **...** is omitted then the loop is executed once for each positional parameter; that is, **in** *$∗* is assumed.

Another example of the use of the **for** loop is the *create* command whose text is

```
for i do >$i; done
```

The command

```
create alpha beta
```

ensures that two empty files *alpha* and *beta* exist and are empty. The notation >*file* may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or newline) is required before **done.**

2.2 Control flow - case

A multiple way branch is provided for by the **case** notation. For example,

```
case $# in
     1)  cat >>$1 ;;
     2)  cat >>$2 <$1 ;;
     *)  echo ´usage: append [ from ] to´ ;;
esac
```

is an *append* command. When called with one argument as

```
append file
```

$# is the string *1* and the standard input is copied onto the end of *file* using the *cat* command.

```
append file1 file2
```

appends the contents of *file1* onto *file2*. If the number of arguments supplied to *append* is other than 1 or 2 then a message is printed indicating proper usage.

The general form of the **case** command is

> **case** *word* **in**
> *pattern* **)** *command-list* **;;**
> **...**
> **esac**

The shell attempts to match *word* with each *pattern,* in the order in which the patterns appear. If a match is found the associated *command-list* is executed and execution of the **case** is complete. Since ∗ is the pattern that matches any string it can be used for the default case.

A word of caution: no check is made to ensure that only one pattern matches the case argument. The first match found defines the set of commands to be executed. In the example below the commands following the second ∗ will never be executed.

```
case $# in
      *) ... ;;
      *) ... ;;
esac
```

Another example of the use of the **case** construction is to distinguish between different forms of an argument. The following example is a fragment of a *cc* command.

```
for i
do case $i in
   −[ocs])        ... ;;
   −*)   echo ´unknown flag $i´ ;;
   *.c)  /lib/c0 $i ... ;;
   *)    echo ´unexpected argument $i´ ;;
   esac
done
```

To allow the same commands to be associated with more than one pattern the **case** command provides for alternative patterns separated by a | . For example,

```
case $i in
      −x|−y)      ...
esac
```

is equivalent to

```
case $i in
      −[xy])      ...
esac
```

The usual quoting conventions apply so that

```
case $i in
      \?) ...
```

will match the character **?**.

2.3 Here documents

The shell procedure *tel* in section 2.1 uses the file **/usr/lib/telnos** to supply the data for *grep.* An alternative is to include this data within the shell procedure as a *here* document, as in,

```
for i
do grep $i ≪!
   ...
   fred mh0123
   bert mh0789
   ...
!
done
```

In this example the shell takes the lines between ≪**!** and **!** as the standard input for *grep.* The string **!** is arbitrary, the document being terminated by a line that consists of the string following ≪ .

Parameters are substituted in the document before it is made available to *grep* as illustrated by the following procedure called *edg*.

```
ed $3 ≪%
g/$1/s//$2/g
w
%
```

The call

```
edg string1 string2 file
```

is then equivalent to the command

```
ed file ≪%
g/string1/s//string2/g
w
%
```

and changes all occurrences of *string1* in *file* to *string2*. Substitution can be prevented using \ to quote the special character **$** as in

```
ed $3 ≪+
1,\$s/$1/$2/g
w
+
```

(This version of *edg* is equivalent to the first except that *ed* will print a **?** if there are no occurrences of the string **$1**.) Substitution within a *here* document may be prevented entirely by quoting the terminating string, for example,

```
grep $i ≪\#
...
#
```

The document is presented without modification to *grep.* If parameter substitution is not required in a *here* document this latter form is more efficient.

2.4 Shell variables

The shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits and underscores. Variables may be given values by writing, for example,

```
user=fred box=m000 acct=mh0000
```

which assigns values to the variables **user, box** and **acct.** A variable may be set to the null string by saying, for example,

```
null=
```

The value of a variable is substituted by preceding its name with **$**; for example,

```
echo $user
```

will echo *fred.*

Variables may be used interactively to provide abbreviations for frequently used strings. For example,

```
b=/usr/fred/bin
mv pgm $b
```

will move the file *pgm* from the current directory to the directory **/usr/fred/bin**. A more general notation is available for parameter (or variable) substitution, as in,

```
echo ${user}
```

which is equivalent to

```
echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps
ps a >${tmp}a
```

will direct the output of *ps* to the file **/tmp/psa,** whereas,

```
ps a >$tmpa
```

would cause the value of the variable **tmpa** to be substituted.

Except for **$?** the following are set initially by the shell. **$?** is set after executing each command.

$? The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully, otherwise a non-zero exit status is returned. Testing the value of return codes is dealt with later under **if** and **while** commands.

$# The number of positional parameters (in decimal). Used, for example, in the *append* command to check the number of parameters.

$$ The process number of this shell (in decimal). Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example,

```
ps a >/tmp/ps$$
...
rm /tmp/ps$$
```

$! The process number of the last process run in the background (in decimal).

$− The current shell flags, such as **−x** and **−v**.

Some variables have a special meaning to the shell and should be avoided for general use.

$MAIL When used interactively the shell looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at the shell prints the message *you have mail* before prompting for the next command. This variable is typically set in the file **.profile,** in the user's login directory. For example,

```
MAIL=/usr/mail/fred
```

$HOME The default argument for the *cd* command. The current directory is used to resolve file name references that do not begin with a **/**, and is changed using the *cd* command. For example,

```
cd /usr/fred/bin
```

makes the current directory **/usr/fred/bin**.

```
cat wn
```

will print on the terminal the file *wn* in this directory. The command *cd* with no argument is equivalent to

```
cd $HOME
```

This variable is also typically set in the the user's login profile.

$PATH A list of directories that contain commands (the *search path*). Each time a command is executed by the shell a list of directories is searched for an executable file. If **$PATH** is not set then the current directory, **/bin**, and **/usr/bin** are searched by default. Otherwise **$PATH** consists of directory names separated by **:**. For example,

PATH=**:**/usr/fred/bin**:**/bin**:**/usr/bin

specifies that the current directory (the null string before the first **:**), **/usr/fred/bin, /bin** and **/usr/bin** are to be searched in that order. In this way individual users can have their own 'private' commands that are accessible independently of the current directory. If the command name contains a **/** then this directory search is not used; a single attempt is made to execute the command.

$PS1 The primary shell prompt string, by default, '**$** '.

$PS2 The shell prompt when further input is needed, by default, '> '.

$IFS The set of characters used by *blank interpretation* (see section 3.4).

2.5 The test command

The *test* command, although not part of the shell, is intended for use by shell programs. For example,

```
test −f file
```

returns zero exit status if *file* exists and non-zero exit status otherwise. In general *test* evaluates a predicate and returns the result as its exit status. Some of the more frequently used *test* arguments are given here, see *test* (1) for a complete specification.

test s	true if the argument *s* is not the null string
test −f file	true if *file* exists
test −r file	true if *file* is readable
test −w file	true if *file* is writable
test −d file	true if *file* is a directory

2.6 Control flow - while

The actions of the **for** loop and the **case** branch are determined by data available to the shell. A **while** or **until** loop and an **if then else** branch are also provided whose actions are determined by the exit status returned by commands. A **while** loop has the general form

while *command-list*$_1$
do *command-list*$_2$
done

The value tested by the **while** command is the exit status of the last simple command following **while.** Each time round the loop *command-list*$_1$ is executed; if a zero exit status is returned then *command-list*$_2$ is executed; otherwise, the loop terminates. For example,

```
while test $1
do ...
    shift
done
```

is equivalent to

```
for i
do ...
done
```

shift is a shell command that renames the positional parameters **$2, $3, . . .** as **$1, $2, . . .** and loses **$1.**

Another kind of use for the **while/until** loop is to wait until some external event occurs and then run some commands. In an **until** loop the termination condition is reversed. For example,

```
until test −f file
do sleep 300; done
commands
```

will loop until *file* exists. Each time round the loop it waits for 5 minutes before trying again. (Presumably another process will eventually create the file.)

2.7 Control flow - if

Also available is a general conditional branch of the form,

```
if command-list
then    command-list
else    command-list
fi
```

that tests the value returned by the last simple command following **if.**

The **if** command may be used in conjunction with the *test* command to test for the existence of a file as in

```
if test −f file
then    process file
else    do something else
fi
```

An example of the use of **if, case** and **for** constructions is given in section 2.10 .

A multiple test **if** command of the form

```
if ...
then    ...
else    if ...
        then    ...
        else    if ...
                ...
                fi
        fi
fi
```

may be written using an extension of the **if** notation as,

```
if ...
then    ...
elif    ...
then    ...
elif    ...
...
fi
```

The following example is the *touch* command which changes the 'last modified' time for a list of files. The command may be used in conjunction with *make* (1) to force recompilation of a list of files.

```
flag=
for i
do case $i in
    −c)  flag=N ;;
    ∗)   if test −f $i
         then    ln $i junk$$; rm junk$$
         elif test $flag
         then    echo file \´$i\´ does not exist
         else    >$i
         fi
     esac
done
```

The **−c** flag is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable *flag* is set to some non-null string if the **−c** argument is encountered. The commands

```
ln ...; rm ...
```

make a link to the file and then remove it thus causing the last modified date to be updated.

The sequence

```
if command1
then    command2
fi
```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes *command2* only if *command1* fails. In each case the value returned is that of the last simple command executed.

2.8 Command grouping

Commands may be grouped in two ways,

{ *command-list* **;** }

and

(*command-list* **)**

In the first *command-list* is simply executed. The second form executes *command-list* as a separate process. For example,

```
(cd x; rm junk )
```

executes *rm junk* in the directory **x** without changing the current directory of the invoking shell.

The commands

```
cd x; rm junk
```

have the same effect but leave the invoking shell in the directory **x.**

2.9 Debugging shell procedures

The shell provides two tracing mechanisms to help when debugging shell procedures. The first is invoked within the procedure as

set −v

(**v** for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by saying

sh −v proc **...**

where *proc* is the name of the shell procedure. This flag may be used in conjunction with the −**n** flag which prevents execution of subsequent commands. (Note that saying *set −n* at a terminal will render the terminal useless until an end-of-file is typed.)

The command

set −x

will produce an execution trace. Following parameter substitution each command is printed as it is executed. (Try these at the terminal to see what effect they have.) Both flags may be turned off by saying

set −

and the current setting of the shell flags is available as **$−.**

2.10 The man command

The following is the *man* command which is used to print sections of the UNIX manual. It is called, for example, as

man sh
man −t ed
man 2 fork

In the first the manual section for *sh* is printed. Since no section is specified, section 1 is used. The second example will typeset (−**t** option) the manual section for *ed.* The last prints the *fork* manual page from section 2.

```
cd /usr/man

: ´colon is the comment command´
: ´default is nroff ($N), section 1 ($s)´
N=n s=1

for i
do case $i in

    [1−9]∗)       s=$i ;;

    −t)   N=t ;;

    −n)   N=n ;;

    −∗)   echo unknown flag \´$i\´ ;;

    ∗)    if test −f man$s/$i.$s
          then    ${N}roff man0/${N}aa man$s/$i.$s
          else    : ´look through all manual sections´
                  found=no
                  for j in 1 2 3 4 5 6 7 8 9
                  do if test −f man$j/$i.$j
                     then man $j $i
                          found=yes
                     fi
                  done
                  case $found in
                       no) echo ´$i: manual page not found´
                  esac
          fi
    esac
done
```

Figure 1. A version of the man command

3.0 Keyword parameters

Shell variables may be given values by assignment or when a shell procedure is invoked. An argument to a shell procedure of the form *name=value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking shell is not affected. For example,

    user=fred command

will execute *command* with **user** set to *fred*. The **−k** flag causes arguments of the form *name=value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain they are available as positional parameters **$1, $2, ...**

The *set* command may also be used to set positional parameters from within a procedure. For example,

    set − *

will set **$1** to the first file name in the current directory, **$2** to the next, and so on. Note that the first argument, −, ensures correct treatment when the first file name begins with a −.

3.1 Parameter transmission

When a shell procedure is invoked both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a shell procedure by specifying in advance that such parameters are to be exported. For example,

    export user box

marks the variables **user** and **box** for export. When a shell procedure is invoked copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking shell. It is generally true of a shell procedure that it may not modify the state of its caller without explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared *readonly*. The form of this command is the same as that of the *export* command,

    readonly name ...

Subsequent attempts to set readonly variables are illegal.

3.2 Parameter substitution

If a shell parameter is not set then the null string is substituted for it. For example, if the variable **d** is not set

    echo $d

or

    echo ${d}

will echo nothing. A default string may be given as in

    echo ${d−.}

which will echo the value of the variable **d** if it is set and '**.**' otherwise. The default string is evaluated using the usual quoting conventions so that

    echo ${d−´*´}

will echo * if the variable **d** is not set. Similarly

    echo ${d−$1}

will echo the value of **d** if it is set and the value (if any) of **$1** otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.}
```

which substitutes the same string as

```
echo ${d–.}
```

and if **d** were not previously set then it will be set to the string '**.**'. (The notation ${**...=...**} is not available for positional parameters.)

If there is no sensible default then the notation

```
echo ${d?message}
```

will echo the value of the variable **d** if it has one, otherwise *message* is printed by the shell and execution of the shell procedure is abandoned. If *message* is absent then a standard message is printed. A shell procedure that requires some parameters to be set might start as follows.

```
: ${user?} ${acct?} ${bin?}
...
```

Colon (**:**) is a command that is built in to the shell and does nothing once its arguments have been evaluated. If any of the variables **user, acct** or **bin** are not set then the shell will abandon execution of the procedure.

3.3 Command substitution

The standard output from a command can be substituted in a similar way to parameters. The command *pwd* prints on its standard output the name of the current directory. For example, if the current directory is **/usr/fred/bin** then the command

```
d=`pwd`
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between grave accents (`**...**`) is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ` must be escaped using a **\\.** For example,

```
ls `echo "$1"`
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including *here* documents) and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within shell procedures. An example of such a command is *basename* which removes a specified suffix from a string. For example,

```
basename main.c .c
```

will print the string *main*. Its use is illustrated by the following fragment from a *cc* command.

```
case $A in
    ...
    *.c)        B=`basename $A .c`
    ...
esac
```

that sets **B** to the part of **$A** with the suffix **.c** stripped.

Here are some composite examples.

- **for i in \`ls −t\`; do ...**
  The variable **i** is set to the names of files in time order, most recent first.
- **set \`date\`; echo $6 $2 $3, $4**
  will print, e.g., *1977 Nov 1, 23:59:59*

3.4 Evaluation and quoting

The shell is a macro processor that provides parameter substitution, command substitution and file name generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in appendix A. Before a command is executed the following substitutions occur.

- parameter substitution, e.g. **$user**
- command substitution, e.g. **\`pwd\`**

  Only one evaluation occurs so that if, for example, the value of the variable **X** is the string *$y* then

  ```
  echo $X
  ```

  will echo *$y*.
- blank interpretation

  Following the above substitutions the resulting characters are broken into non-blank words (*blank interpretation*). For this purpose 'blanks' are the characters of the string **$IFS**. By default, this string consists of blank, tab and newline. The null string is not regarded as a word unless it is quoted. For example,

  ```
  echo ´´
  ```

  will pass on the null string as the first argument to *echo*, whereas

  ```
  echo $null
  ```

  will call *echo* with no arguments if the variable **null** is not set or set to the null string.
- file name generation

  Each word is then scanned for the file pattern characters ***, ?** and **[...]** and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a **for** loop. Only substitution occurs in the *word* used for a **case** branch.

As well as the quoting mechanisms described earlier using \ and **´...´** a third quoting mechanism is provided using double quotes. Within double quotes parameter and command substitution occurs but file name generation and the interpretation of blanks does not. The following characters have a special meaning within double quotes and may be quoted using **\\.**

$	parameter substitution
\`	command substitution
"	ends the quoted string
\	quotes the special characters **$** \` " \

For example,

```
echo "$x"
```

will pass the value of the variable **x** as a single argument to *echo.* Similarly,

```
echo "$*"
```

will pass the positional parameters as a single argument and is equivalent to echo "$1 $2 ..."

The notation **$@** is the same as **$*** except when it is quoted.

echo "$@"

will pass the positional parameters, unevaluated, to *echo* and is equivalent to

echo "$1" "$2" ...

The following table gives, for each quoting mechanism, the shell metacharacters that are evaluated.

	metacharacter					
	\	$	*	`	"	´
´	n	n	n	n	n	t
`	y	n	n	t	n	n
"	y	y	n	y	t	n

t	terminator
y	interpreted
n	not interpreted

Figure 2. Quoting mechanisms

In cases where more than one evaluation of a string is required the built-in command *eval* may be used. For example, if the variable **X** has the value *$y*, and if **y** has the value *pqr* then

eval echo $X

will echo the string *pqr*.

In general the *eval* command evaluates its arguments (as do all commands) and treats the result as input to the shell. The input is read and the resulting command(s) executed. For example,

```
wg=´eval who|grep´
$wg fred
```

is equivalent to

who|grep fred

In this example, *eval* is required since there is no interpretation of metacharacters, such as **|, following substitution.**

3.5 Error handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a terminal (as determined by *gtty* (2)). A shell invoked with the **–i** flag is also interactive.

Execution of a command (see also 3.7) may fail for any of the following reasons.

- Input output redirection may fail. For example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a "bus error" or "memory fault". See Figure 2 below for a complete list of UNIX signals.
- The command terminates normally but returns a non-zero exit status.

In all of these cases the shell will go on to execute the next command. Except for the last case an error message will be printed by the shell. All remaining errors cause the shell to exit from a command procedure. An interactive shell will return to read another command from the terminal. Such errors include the following.

- Syntax errors. e.g., if **...** then **...** done
- A signal such as interrupt. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as *cd.*

The shell flag **−e** causes the shell to terminate if any error is detected.

1	hangup
2	interrupt
3*	quit
4*	illegal instruction
5*	trace trap
6*	IOT instruction
7*	EMT instruction
8*	floating point exception
9	kill (cannot be caught or ignored)
10*	bus error
11*	segmentation violation
12*	bad argument to system call
13	write on a pipe with no one to read it
14	alarm clock
15	software termination (from *kill* (1))

Figure 3. UNIX signals

Those signals marked with an asterisk produce a core dump if not caught. However, the shell itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to shell programs are 1, 2, 3, 14 and 15.

3.6 Fault handling

Shell procedures normally terminate when an interrupt is received from the terminal. The *trap* command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap ´rm /tmp/ps$$; exit´ 2
```

sets a trap for signal 2 (terminal interrupt), and if this signal is received will execute the commands

```
rm /tmp/ps$$; exit
```

exit is another built-in command that terminates execution of a shell procedure. The *exit* is required; otherwise, after the trap has been taken, the shell will resume executing the procedure at the place where it was interrupted.

UNIX signals can be handled in one of three ways. They can be ignored, in which case the signal is never sent to the process. They can be caught, in which case the process must decide what action to take when the signal is received. Lastly, they can be left to cause termination of the process without it having to take any further action. If a signal is being ignored on entry to the shell procedure, for example, by invoking it in the background (see 3.7) then *trap* commands (and the signal) are ignored.

The use of *trap* is illustrated by this modified version of the *touch* command (Figure 4). The cleanup action is to remove the file **junk$$**.

```
flag=
trap ´rm −f junk$$; exit´ 1 2 3 15
for i
do case $i in
    −c)  flag=N ;;
    ∗)   if test −f $i
         then    ln $i junk$$; rm junk$$
         elif test $flag
         then    echo file \´$i\´ does not exist
         else    >$i
         fi
    esac
done
```

Figure 4. The touch command

The *trap* command appears before the creation of the temporary file; otherwise it would be possible for the process to die without removing the file.

Since there is no signal 0 in UNIX it is used by the shell to indicate the commands to be executed on exit from the shell procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to trap. The following fragment is taken from the *nohup* command.

```
trap ´´ 1 2 3 15
```

which causes *hangup, interrupt, quit* and *kill* to be ignored both by the procedure and by invoked commands.

Traps may be reset by saying

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```

The procedure *scan* (Figure 5) is an example of the use of *trap* where there is no exit in the trap command. *scan* takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when *scan* is waiting for input.

```
d=`pwd`
for i in ∗
do if test −d $d/$i
    then cd $d/$i
         while echo "$i:"
                trap exit 2
                read x
         do trap : 2; eval $x; done
    fi
done
```

Figure 5. The scan command

read x is a built-in command that reads one line from the standard input and places the result in the variable **x**. It returns a non-zero exit status if either an end-of-file is read or an interrupt is received.

3.7 Command execution

To run a command (other than a built-in) the shell first creates a new process using the system call *fork*. The execution environment for the command includes input, output and the states of signals, and is established in the child process before the command is executed. The built-in command *exec* is used in the rare cases when no fork is required and simply replaces the shell with a new command. For example, a simple version of the *nohup* command looks like

```
trap ´´ 1 2 3 15
exec $*
```

The *trap* turns off the signals specified so that they are ignored by subsequently created commands and *exec* replaces the shell by the command specified.

Most forms of input output redirection have already been described. In the following *word* is only subject to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo ... >*.c
```

will write its output into a file whose name is ***.c**. Input output specifications are evaluated left to right as they appear in the command.

> *word* — The standard output (file descriptor 1) is sent to the file *word* which is created if it does not already exist.

≫ *word* — The standard output is sent to file *word*. If the file exists then output is appended (by seeking to the end); otherwise the file is created.

< *word* — The standard input (file descriptor 0) is taken from the file *word*.

≪ *word* — The standard input is taken from the lines of shell input that follow up to but not including a line consisting only of *word*. If *word* is quoted then no interpretation of the document occurs. If *word* is not quoted then parameter and command substitution occur and \ is used to quote the characters \ **$** ` and the first character of *word*. In the latter case **\newline** is ignored (c.f. quoted strings).

>& *digit* — The file descriptor *digit* is duplicated using the system call *dup* (2) and the result is used as the standard output.

<& *digit* — The standard input is duplicated from file descriptor *digit*.

<&− — The standard input is closed.

>&− — The standard output is closed.

Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to *file*.

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking file descriptor 2 is created by duplicating file descriptor 1 but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. Firstly, the default standard input for such a command is the empty file **/dev/null**. This prevents two processes (the shell and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example, ed file &

would allow both the editor and the shell to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason the UNIX convention for a signal is that if it is set to 1 (ignored) then it is never changed even for a short time. Note that the shell command *trap* has no effect for an ignored signal.

3.8 Invoking the shell

The following flags are interpreted by the shell when it is invoked. If the first character of argument zero is a minus, then commands are read from the file **.profile.**

–c *string*
If the **–c** flag is present then commands are read from *string*.

–s If the **–s** flag is present or if no arguments remain then commands are read from the standard input. Shell output is written to file descriptor 2.

–i If the **–i** flag is present or if the shell input and output are attached to a terminal (as told by *gtty*) then this shell is *interactive.* In this case TERMINATE is ignored (so that **kill 0** does not kill an interactive shell) and INTERRUPT is caught and ignored (so that **wait** is interruptable). In all cases QUIT is ignored by the shell.

Acknowledgements

The design of the shell is based in part on the original UNIX shell[3] and the PWB/UNIX shell,[4] some features having been taken from both. Similarities also exist with the command interpreters of the Cambridge Multiple Access System[5] and of CTSS.[6]

I would like to thank Dennis Ritchie and John Mashey for many discussions during the design of the shell. I am also grateful to the members of the Computing Science Research Center and to Joe Maranzano for their comments on drafts of this document.

References

1. B. W. Kernighan, *UNIX for Beginners*, 1978.
2. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories (1978). Seventh Edition.
3. K. Thompson, ''The UNIX Command Language,'' pp. 375-384 in *Structured Programming—Infotech State of the Art Report*, Infotech International Ltd., Nicholson House, Maidenhead, Berkshire, England (March 1975).
4. J. R. Mashey, *PWB/UNIX Shell Tutorial*, September 30, 1977.
5. D. F. Hartley (Ed.), *The Cambridge Multiple Access System – Users Reference Manual,* University Mathematical Laboratory, Cambridge, England (1968).
6. P. A. Crisman (Ed.), *The Compatible Time-Sharing System,* M.I.T. Press, Cambridge, Mass. (1965).

Appendix A - Grammar

item: *word*
input-output
name = value

simple-command: *item*
simple-command item

command: *simple-command*
(*command-list* **)**
{ *command-list* }
for *name* **do** *command-list* **done**
for *name* **in** *word* **...** **do** *command-list* **done**
while *command-list* **do** *command-list* **done**
until *command-list* **do** *command-list* **done**
case *word* **in** *case-part* **...** **esac**
if *command-list* **then** *command-list else-part* **fi**

pipeline: *command*
pipeline **|** *command*

andor: *pipeline*
andor **&&** *pipeline*
andor || *pipeline*

command-list: *andor*
command-list **;**
command-list **&**
command-list **;** *andor*
command-list **&** *andor*

input-output: > *file*
< *file*
≫ *word*
≪ *word*

file: *word*
& *digit*
& −

case-part: *pattern*) *command-list* **;;**

pattern: *word*
pattern | *word*

else-part: **elif** *command-list* **then** *command-list else-part*
else *command-list*
empty

empty:

word: a sequence of non-blank characters

name: a sequence of letters, digits or underscores starting with a letter

digit: **0 1 2 3 4 5 6 7 8 9**

Appendix B - Meta-characters and Reserved Words

a) syntactic

- **|** pipe symbol
- **&&** ‘andf’ symbol
- **||** ‘orf’ symbol
- **;** command separator
- **;;** case delimiter
- **&** background commands
- **( )** command grouping
- **<** input redirection
- **<<** input from a here document
- **>** output creation
- **>>** output append

b) patterns

- ***** match any character(s) including none
- **?** match any single character
- **[...]** match any of the enclosed characters

c) substitution

- **${...}** substitute shell variable
- **\`...\`** substitute command output

d) quoting

- **** quote the next character
- **´...´** quote the enclosed characters except for ´
- **"..."** quote the enclosed characters except for **$** \` \\ "

e) reserved words

if then else elif fi
case in esac
for while until do done
{ }

LEARN — Computer-Aided Instruction on UNIX (Second Edition)

Brian W. Kernighan

Michael E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the second version of the *learn* program for interpreting CAI scripts on the UNIX† operating system, and a set of scripts that provide a computerized introduction to the system.

Six current scripts cover basic commands and file handling, the editor, additional file handling commands, the *eqn* program for mathematical typing, the ''−ms'' package of formatting macros, and an introduction to the C programming language. These scripts now include a total of about 530 lessons.

Many users from a wide variety of backgrounds have used *learn* to acquire basic UNIX skills. Most usage involves the first two scripts, an introduction to UNIX files and commands, and the UNIX editor.

The second version of *learn* is about four times faster than the previous one in CPU utilization, and much faster in perceived time because of better overlap of computing and printing. It also requires less file space than the first version. Many of the lessons have been revised; new material has been added to reflect changes and enhancements in UNIX itself. Script-writing is also easier because of revisions to the script language.

January 30, 1979

†UNIX is a Trademark of Bell Laboratories.

LEARN — Computer-Aided Instruction on UNIX
(Second Edition)

Brian W. Kernighan

Michael E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

1. Educational Assumptions and Design.

First, the way to teach people how to do something is to have them do it. Scripts should not contain long pieces of explanation; they should instead frequently ask the student to do some task. So teaching is always by example: the typical script fragment shows a small example of some technique and then asks the user to either repeat that example or produce a variation on it. All are intended to be easy enough that most students will get most questions right, reinforcing the desired behavior.

Most lessons fall into one of three types. The simplest presents a lesson and asks for a yes or no answer to a question. The student is given a chance to experiment before replying. The script checks for the correct reply. Problems of this form are sparingly used.

The second type asks for a word or number as an answer. For example a lesson on files might say

> *How many files are there in the current directory? Type ''answer N'', where N is the number of files.*

The student is expected to respond (perhaps after experimenting) with

> *answer 17*

or whatever. Surprisingly often, however, the idea of a substitutable argument (i.e., replacing *N* by 17) is difficult for non-programmer students, so the first few such lessons need real care.

The third type of lesson is open-ended — a task is set for the student, appropriate parts of the input or output are monitored, and the student types *ready* when the task is done. Figure 1 shows a sample dialog that illustrates the last of these, using two lessons about the *cat* (concatenate, i.e., print) command taken from early in the script that teaches file handling. Most *learn* lessons are of this form.

After each correct response the computer congratulates the student and indicates the lesson number that has just been completed, permitting the student to restart the script after that lesson. If the answer is wrong, the student is offered a chance to repeat the lesson. The ''speed'' rating of the student (explained in section 5) is given after the lesson number when the lesson is completed successfully; it is printed only for the aid of script authors checking out possible errors in the lessons.

It is assumed that there is no foolproof way to determine if the student truly ''understands'' what he or she is doing; accordingly, the current *learn* scripts only measure performance, not comprehension. If the student can perform a given task, that is deemed to be ''learning.''[1]

The main point of using the computer is that what the student does is checked for correctness immediately. Unlike many CAI scripts, however, these scripts provide few facilities for dealing with wrong answers. In practice, if most of the answers are not right the script is a failure; the universal solution to student error is to provide a new, easier script. Anticipating possible wrong answers is an endless job, and it is really easier as well as better to provide a simpler script.

Along with this goes the assumption that anything can be taught to anybody if it can be broken into sufficiently small pieces. Anything not absorbed in a single chunk is just subdivided.

Figure 1: Sample dialog from basic files script

(Student responses in italics; '$' is the prompt)

A file can be printed on your terminal
by using the "cat" command. Just say
"cat file" where "file" is the file name.
For example, there is a file named
"food" in this directory. List it
by saying "cat food"; then type "ready".
$ *cat food*
  this is the file
  named food.
$ *ready*

Good. Lesson 3.3a (1)

Of course, you can print any file with "cat".
In particular, it is common to first use
"ls" to find the name of a file and then "cat"
to print it. Note the difference between
"ls", which tells you the name of the file,
and "cat", which tells you the contents.
One file in the current directory is named for
a President. Print the file, then type "ready".
$ *cat President*
cat: can't open President
$ *ready*

Sorry, that's not right. Do you want to try again? *yes*
Try the problem again.
$ *ls*
.ocopy
X1
roosevelt
$ *cat roosevelt*
  this file is named roosevelt
  and contains three lines of
  text.
$ *ready*

Good. Lesson 3.3b (0)

The "cat" command can also print several files
at once. In fact, it is named "cat" as an abbreviation
for "concatenate"....

To avoid boring the faster students, however, an effort is made in the files and editor scripts to provide three tracks of different difficulty. The fastest sequence of lessons is aimed at roughly the bulk and speed of a typical tutorial manual and should be adequate for review and for well-prepared students. The next track is intended for most users and is roughly twice as long. Typically, for example, the fast track might present an idea and ask for a variation on the example shown; the normal track will first ask the student to repeat the example that was shown before attempting a variation. The third and slowest track, which is often three or four times the length of the fast track, is intended to be adequate for anyone. (The lessons of Figure 1 are from the third track.) The multiple tracks also mean that a student repeating a course is unlikely to hit the same series of lessons; this makes it profitable for a shaky user to back up and try again, and many students have done so.

The tracks are not completely distinct, however. Depending on the number of correct answers the student has given for the last few lessons, the program may switch tracks. The driver is actually capable of following an arbitrary directed graph of lesson sequences, as discussed in section 5. Some more structured arrangement, however, is used in all current scripts to aid the script writer in organizing the material into lessons. It is sufficiently difficult to write lessons that the three-track theory is not followed very closely except in the files and editor scripts. Accordingly, in some cases, the fast track is produced merely by skipping lessons from the slower track. In others, there is essentially only one track.

The main reason for using the *learn* program rather than simply writing the same material as a workbook is not the selection of tracks, but actual hands-on experience. Learning by doing is much more effective than pencil and paper exercises.

Learn also provides a mechanical check on performance. The first version in fact would not let the student proceed unless it received correct answers to the questions it set and it would not tell a student the right answer. This somewhat Draconian approach has been moderated in version 2. Lessons are sometimes badly worded or even just plain wrong; in such cases, the student has no recourse. But if a student is simply unable to complete one lesson, that should not prevent access to the rest. Accordingly, the current version of *learn* allows the student to skip a lesson that he cannot pass; a ‘‘no’’ answer to the ‘‘Do you want to try again?’’ question in Figure 1 will pass to the next lesson. It is still true that *learn* will not tell the student the right answer.

Of course, there are valid objections to the assumptions above. In particular, some students may object to not understanding what they are doing; and the procedure of smashing everything into small pieces may provoke the retort ‘‘you can’t cross a ditch in two jumps.’’ Since writing CAI scripts is considerably more tedious than ordinary manuals, however, it is safe to assume that there will always be alternatives to the scripts as a way of learning. In fact, for a reference manual of 3 or 4 pages it would not be surprising to have a tutorial manual of 20 pages and a (multi-track) script of 100 pages. Thus the reference manual will exist long before the scripts.

2. Scripts.

As mentioned above, the present scripts try at most to follow a three-track theory. Thus little of the potential complexity of the possible directed graph is employed, since care must be taken in lesson construction to see that every necessary fact is presented in every possible path through the units. In addition, it is desirable that every unit have alternate successors to deal with student errors.

In most existing courses, the first few lessons are devoted to checking prerequisites. For example, before the student is allowed to proceed through the editor script the script verifies that the student understands files and is able to type. It is felt that the sooner lack of student preparation is detected, the easier it will be on the student. Anyone proceeding through the scripts should be getting mostly correct answers; otherwise, the system will be unsatisfactory both because the wrong habits are being learned and because the scripts make little effort to deal with wrong answers. Unprepared students should not be encouraged to continue with scripts.

There are some preliminary items which the student must know before any scripts can be tried. In particular, the student must know how to connect to a UNIX† system, set the terminal properly, log in, and execute simple commands (e.g., *learn* itself). In addition, the character erase and line kill conventions (# and @) should be known. It is hard to see how this much could be taught by computer-aided instruction, since a student who does not know these basic skills will not be able to run the learning program. A brief description on paper is provided (see Appendix A), although assistance will be needed for the first few minutes. This assistance, however, need not be highly skilled.

†UNIX is a Trademark of Bell Laboratories.

The first script in the current set deals with files. It assumes the basic knowledge above and teaches the student about the *ls*, *cat*, *mv*, *rm*, *cp* and *diff* commands. It also deals with the abbreviation characters *, ?, and [ ] in file names. It does not cover pipes or I/O redirection, nor does it present the many options on the *ls* command.

This script contains 31 lessons in the fast track; two are intended as prerequisite checks, seven are review exercises. There are a total of 75 lessons in all three tracks, and the instructional passages typed at the student to begin each lesson total 4,476 words. The average lesson thus begins with a 60-word message. In general, the fast track lessons have somewhat longer introductions, and the slow tracks somewhat shorter ones. The longest message is 144 words and the shortest 14.

The second script trains students in the use of the UNIX context editor *ed*, a sophisticated editor using regular expressions for searching.[2] All editor features except encryption, mark names and ';' in addressing are covered. The fast track contains 2 prerequisite checks, 93 lessons, and a review lesson. It is supplemented by 146 additional lessons in other tracks.

A comparison of sizes may be of interest. The *ed* description in the reference manual is 2,572 words long. The *ed* tutorial[3] is 6,138 words long. The fast track through the *ed* script is 7,407 words of explanatory messages, and the total *ed* script, 242 lessons, has 15,615 words. The average *ed* lesson is thus also about 60 words; the largest is 171 words and the smallest 10. The original *ed* script represents about three man-weeks of effort.

The advanced file handling script deals with *ls* options, I/O diversion, pipes, and supporting programs like *pr*, *wc*, *tail*, *spell* and *grep*. (The basic file handling script is a prerequisite.) It is not as refined as the first two scripts; this is reflected at least partly in the fact that it provides much less of a full three-track sequence than they do. On the other hand, since it is perceived as ''advanced,'' it is hoped that the student will have somewhat more sophistication and be better able to cope with it at a reasonably high level of performance.

A fourth script covers the *eqn* language for typing mathematics. This script must be run on a terminal capable of printing mathematics, for instance the DASI 300 and similar Diablo-based terminals, or the nearly extinct Model 37 teletype. Again, this script is relatively short of tracks: of 76 lessons, only 17 are in the second track and 2 in the third track. Most of these provide additional practice for students who are having trouble in the first track.

The −*ms* script for formatting macros is a short one-track only script. The macro package it describes is no longer the standard, so this script will undoubtedly be superseded in the future. Furthermore, the linear style of a single learn script is somewhat inappropriate for the macros, since the macro package is composed of many independent features, and few users need all of them. It would be better to have a selection of short lesson sequences dealing with the features independently.

The script on C is in a state of transition. It was originally designed to follow a tutorial on C, but that document has since become obsolete. The current script has been partially converted to follow the order of presentation in *The C Programming Language,*[4] but this job is not complete. The C script was never intended to teach C; rather it is supposed to be a series of exercises for which the computer provides checking and (upon success) a suggested solution.

This combination of scripts covers much of the material which any UNIX user will need to know to make effective use of the system. With enlargement of the advanced files course to include more on the command interpreter, there will be a relatively complete introduction to UNIX available via *learn.* Although we make no pretense that *learn* will replace other instructional materials, it should provide a useful supplement to existing tutorials and reference manuals.

3. Experience with Students.

Learn has been installed on many different UNIX systems. Most of the usage is on the first two scripts, so these are more thoroughly debugged and polished. As a (random) sample of user experience, the *learn* program has been used at Bell Labs at Indian Hill for 10,500 lessons in a four month period. About 3600 of these are in the files script, 4100 in the editor, and 1400 in advanced files. The passing rate is about 80%, that is, about 4 lessons are passed for every one failed. There have been 86 distinct users of the files script, and 58 of the editor. On our system at Murray Hill, there have been nearly

2000 lessons over two weeks that include Christmas and New Year. Users have ranged in age from six up.

It is difficult to characterize typical sessions with the scripts; many instances exist of someone doing one or two lessons and then logging out, as do instances of someone pausing in a script for twenty minutes or more. In the earlier version of *learn*, the average session in the files course took 32 minutes and covered 23 lessons. The distribution is quite broad and skewed, however; the longest session was 130 minutes and there were five sessions shorter than five minutes. The average lesson took about 80 seconds. These numbers are roughly typical for non-programmers; a UNIX expert can do the scripts at approximately 30 seconds per lesson, most of which is the system printing.

At present working through a section of the middle of the files script took about 1.4 seconds of processor time per lesson, and a system expert typing quickly took 15 seconds of real time per lesson. A novice would probably take at least a minute. Thus a UNIX system could support ten students working simultaneously with some spare capacity.

4. The Script Interpreter.

The *learn* program itself merely interprets scripts. It provides facilities for the script writer to capture student responses and their effects, and simplifies the job of passing control to and recovering control from the student. This section describes the operation and usage of the driver program, and indicates what is required to produce a new script. Readers only interested in the existing scripts may skip this section.

The file structure used by *learn* is shown in Figure 2. There is one parent directory (named *lib*) containing the script data. Within this directory are subdirectories, one for each subject in which a course is available, one for logging (named *log*), and one in which user sub-directories are created (named *play*). The subject directory contains master copies of all lessons, plus any supporting material for that subject. In a given subdirectory, each lesson is a single text file. Lessons are usually named systematically; the file that contains lesson *n* is called *Ln*.

Figure 2: Directory structure for *learn*

lib			
	play		
		student1	
			files for student1...
		student2	
			files for student2...
	files		
		L0.1a	lessons for files course
		L0.1b	
		...	
	editor		
		...	
	(other courses)		
	log		

When *learn* is executed, it makes a private directory for the user to work in, within the *learn* portion of the file system. A fresh copy of all the files used in each lesson (mostly data for the student to operate upon) is made each time a student starts a lesson, so the script writer may assume that everything is reinitialized each time a lesson is entered. The student directory is deleted after each session; any permanent records must be kept elsewhere.

The script writer must provide certain basic items in each lesson:

(1) the text of the lesson;

(2) the set-up commands to be executed before the user gets control;

(3) the data, if any, which the user is supposed to edit, transform, or otherwise process;

(4) the evaluating commands to be executed after the user has finished the lesson, to decide whether the answer is right; and

(5) a list of possible successor lessons.

Learn tries to minimize the work of bookkeeping and installation, so that most of the effort involved in script production is in planning lessons, writing tutorial paragraphs, and coding tests of student performance.

The basic sequence of events is as follows. First, *learn* creates the working directory. Then, for each lesson, *learn* reads the script for the lesson and processes it a line at a time. The lines in the script are: (1) commands to the script interpreter to print something, to create a files, to test something, etc.; (2) text to be printed or put in a file; (3) other lines, which are sent to the shell to be executed. One line in each lesson turns control over to the user; the user can run any UNIX commands. The user mode terminates when the user types *yes*, *no*, *ready*, or *answer*. At this point, the user's work is tested; if the lesson is passed, a new lesson is selected, and if not the old one is repeated.

Let us illustrate this with the script for the second lesson of Figure 1; this is shown in Figure 3.

Figure 3: Sample Lesson

```
#print
Of course, you can print any file with "cat".
In particular, it is common to first use
"ls" to find the name of a file and then "cat"
to print it. Note the difference between
"ls", which tells you the name of the files,
and "cat", which tells you the contents.
One file in the current directory is named for
a President. Print the file, then type "ready".
#create roosevelt
  this file is named roosevelt
  and contains three lines of
  text.
#copyout
#user
#uncopyout
tail −3 .ocopy >X1
#cmp X1 roosevelt
#log
#next
3.2b 2
```

Lines which begin with # are commands to the *learn* script interpreter. For example,

#print

causes printing of any text that follows, up to the next line that begins with a sharp.

#print file

prints the contents of *file*; it is the same as *cat file* but has less overhead. Both forms of *#print* have the added property that if a lesson is failed, the *#print* will not be executed the second time through; this avoids annoying the student by repeating the preamble to a lesson.

#create filename

creates a file of the specified name, and copies any subsequent text up to a # to the file. This is used for creating and initializing working files and reference data for the lessons.

#user

gives control to the student; each line he or she types is passed to the shell for execution. The *#user* mode is terminated when the student types one of *yes*, *no*, *ready* or *answer*. At that time, the driver resumes interpretation of the script.

#copyin
#uncopyin

Anything the student types between these commands is copied onto a file called *.copy.* This lets the script writer interrogate the student's responses upon regaining control.

#copyout
#uncopyout

Between these commands, any material typed at the student by any program is copied to the file *.ocopy.* This lets the script writer interrogate the effect of what the student typed, which true believers in the performance theory of learning usually prefer to the student's actual input.

#pipe
#unpipe

Normally the student input and the script commands are fed to the UNIX command interpreter (the ''shell'') one line at a time. This won't do if, for example, a sequence of editor commands is provided, since the input to the editor must be handed to the editor, not to the shell. Accordingly, the material between *#pipe* and *#unpipe* commands is fed continuously through a pipe so that such sequences work. If *copyout* is also desired the *copyout* brackets must include the *pipe* brackets.

There are several commands for setting status after the student has attempted the lesson.

#cmp file1 file2

is an in-line implementation of *cmp*, which compares two files for identity.

#match stuff

The last line of the student's input is compared to *stuff*, and the success or fail status is set according to it. Extraneous things like the word *answer* are stripped before the comparison is made. There may be several *#match* lines; this provides a convenient mechanism for handling multiple ''right'' answers. Any text up to a # on subsequent lines after a successful *#match* is printed; this is illustrated in Figure 4, another sample lesson.

#bad stuff

This is similar to *#match*, except that it corresponds to specific failure answers; this can be used to produce hints for particular wrong answers that have been anticipated by the script writer.

#succeed
#fail

print a message upon success or failure (as determined by some previous mechanism).

When the student types one of the ''commands'' *yes*, *no*, *ready*, or *answer*, the driver terminates the *#user* command, and evaluation of the student's work can begin. This can be done either by the built-in commands above, such as *#match* and *#cmp*, or by status returned by normal UNIX commands, typically *grep* and *test*. The last command should return status true (0) if the task was done successfully and false (non-zero) otherwise; this status return tells the driver whether or not the student has successfully passed the lesson.

Performance can be logged:

#log file

Figure 4: Another Sample Lesson

```
#print
What command will move the current line
to the end of the file? Type
"answer COMMAND", where COMMAND is the command.
#copyin
#user
#uncopyin
#match m$
#match .m$
"m$" is easier.
#log
#next
63.1d 10
```

writes the date, lesson, user name and speed rating, and a success/failure indication on *file.* The command

#log

by itself writes the logging information in the logging directory within the *learn* hierarchy, and is the normal form.

#next

is followed by a few lines, each with a successor lesson name and an optional speed rating on it. A typical set might read

```
25.1a   10
25.2a   5
25.3a   2
```

indicating that unit 25.1a is a suitable follow-on lesson for students with a speed rating of 10 units, 25.2a for student with speed near 5, and 25.3a for speed near 2. Speed ratings are maintained for each session with a student; the rating is increased by one each tiee the student gets a lesson right and decreased by four each time the student gets a lesson wrong. Thus the driver tries to maintain a devel such that the users get 80% right answers. The maximum rating is limited to 10 afd the minimum to 0. The initial rating is zero unless the studeft specifies a differeft rating when starting a session.

If the student passes a lesson, a new lesson is sedected and the process repeats. If the student fails, a false status is returned and the program reverts to the previous lesson and tries another alternative. If it can not find another alternative, it skips forward a lesson. *bye*, bye, which causes a graceful exit from the *learn* system. Hanging up is the usual novice's way out.

The lessons may form an arbitrary directed graph, although the present program imposes a limitation on cycles in that it will not present a lesson twice in the same session. If the student is unable to answer one of the exercises correctly, the driver searches for a previous lesson with a set of alternatives as successors (following the *#next* line). From the previous lesson with alternatives one route was taken earlier; the program simply tries a different one.

It is perfectly possible to write sophisticated scripts that evaluate the student's speed of response, or try to estimate the elegance of the answer, or provide detailed analysis of wrong answers. Lesson writing is so tedious already, however, that most of these abilities are likely to go unused.

The driver program depends heavily on features of UNIX that are not available on many other operating systems. These include the ease of manipulating files and directories, file redirection, the ability to use the command interpreter as just another program (even in a pipeline), command status testing and branching, the ability to catch signals like interrupts, and of course the pipeline mechanism itself.

Although some parts of *learn* might be transferable to other systems, some generality will probably be lost.

A bit of history: The first version of *learn* had fewer built-in words in the driver program, and made more use of the facilities of UNIX. For example, file comparison was done by creating a *cmp* process, rather than comparing the two files within *learn*. Lessons were not stored as text files, but as archives. There was no concept of the in-line document; even #*print* had to be followed by a file name. Thus the initialization for each lesson was to extract the archive into the working directory (typically 4-8 files), then #*print* the lesson text.

The combination of such things made *learn* slower. The new version is about 4 or 5 times faster. Furthermore, it appears even faster to the user because in a typical lesson, the printing of the message comes first, and file setup with #*create* can be overlapped with the printng, so that when the program finishes printing, it is really ready for the user to type at it.

It is also a great advantage to the script maintainer that lessons are now just ordinary text files. They can be edited without any difficulty, and UNIX text manipulation tools can be applied to them. The result has been that there is much less resistance to going in and fixing substandard lessons.

5. Conclusions

The following observations can be made about secretaries, typists, and other non-programmers who have used *learn*:

(a) A novice must have assistance with the mechanics of communicating with the computer to get through to the first lesson or two; once the first few lessons are passed people can proceed on their own.

(b) The terminology used in the first few lessons is obscure to those inexperienced with computers. It would help if there were a low level reference card for UNIX to supplement the existing programmer oriented bulky manual and bulky reference card.

(c) The concept of ''substitutable argument'' is hard to grasp, and requires help.

(d) They enjoy the system for the most part. Motivation matters a great deal, however.

It takes an hour or two for a novice to get through the script on file handling. The total time for a reasonably intelligent and motivated novice to proceed from ignorance to a reasonable ability to create new files and manipulate old ones seems to be a few days, with perhaps half of each day spent on the machine.

The normal way of proceeding has been to have students in the same room with someone who knows UNIX and the scripts. Thus the student is not brought to a halt by difficult questions. The burden on the counselor, however, is much lower than that on a teacher of a course. Ideally, the students should be encouraged to proceed with instruction immediately prior to their actual use of the computer. They should exercise the scripts on the same computer and the same kind of terminal that they will later use for their real work, and their first few jobs for the computer should be relatively easy ones. Also, both training and initial work should take place on days when the UNIX hardware and software are working reliably. Rarely is all of this possible, but the closer one comes the better the result. For example, if it is known that the hardware is shaky one day, it is better to attempt to reschedule training for another one. Students are very frustrated by machine downtime; when nothing is happening, it takes some sophistication and experience to distinguish an infinite loop, a slow but functioning program, a program waiting for the user, and a broken machine.*

One disadvantage of training with *learn* is that students come to depend completely on the CAI system, and do not try to read manuals or use other learning aids. This is unfortunate, not only because of the increased demands for completeness and accuracy of the scripts, but because the scripts do not cover all of the UNIX system. New users should have manuals (appropriate for their level) and read them; the scripts ought to be altered to recommend suitable documents and urge students to read them.

* We have even known an expert programmer to decide the computer was broken when he had simply left his terminal in local mode. Novices have great difficulties with such problems.

There are several other difficulties which are clearly evident. From the student's viewpoint, the most serious is that lessons still crop up which simply can't be passed. Sometimes this is due to poor explanations, but just as often it is some error in the lesson itself — a botched setup, a missing file, an invalid test for correctness, or some system facility that doesn't work on the local system in the same way it did on the development system. It takes knowledge and a certain healthy arrogance on the part of the user to recognize that the fault is not his or hers, but the script writer's. Permitting the student to get on with the next lesson regardless does alleviate this somewhat, and the logging facilities make it easy to watch for lessons that no one can pass, but it is still a problem.

The biggest problem with the previous *learn* was speed (or lack thereof) — it was often excruciatingly slow and made a significant drain on the system. The current version so far does not seem to have that difficulty, although some scripts, notably *eqn*, are intrinsically slow. *eqn*, for example, must do a lot of work even to print its introductions, let alone check the student responses, but delay is perceptible in all scripts from time to time.

Another potential problem is that it is possible to break *learn* inadvertently, by pushing interrupt at the wrong time, or by removing critical files, or any number of similar slips. The defenses against such problems have steadily been improved, to the point where most students should not notice difficulties. Of course, it will always be possible to break *learn* maliciously, but this is not likely to be a problem.

One area is more fundamental — some UNIX commands are sufficiently global in their effect that *learn* currently does not allow them to be executed at all. The most obvious is *cd*, which changes to another directory. The prospect of a student who is learning about directories inadvertently moving to some random directory and removing files has deterred us from even writing lessons on *cd*, but ultimately lessons on such topics probably should be added.

6. Acknowledgments

We are grateful to all those who have tried *learn,* for we have benefited greatly from their suggestions and criticisms. In particular, M. E. Bittrich, J. L. Blue, S. I. Feldman, P. A. Fox, and M. J. McAlpin have provided substantial feedback. Conversations with E. Z. Rothkopf also provided many of the ideas in the system. We are also indebted to Don Jackowski for serving as a guinea pig for the second version, and to Tom Plum for his efforts to improve the C script.

References

1. B. F. Skinner, ''Why We Need Teaching Machines,'' *Harvard Educational Review* **31**, pp.377-398 (1961).
2. K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories (May 1975). See section *ed* (I).
3. B. W. Kernighan, *A Tutorial Introduction to the Unix Editor ed*, 1974.
4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice Hall (1978).

Typing Documents on the UNIX System: Using the –ms Macros with Troff and Nroff

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This document describes a set of easy-to-use macros for preparing documents on the UNIX system. Documents may be produced on either the phototypesetter or a on a computer terminal, without changing the input.

The macros provide facilities for paragraphs, sections (optionally with automatic numbering), page titles, footnotes, equations, tables, two-column format, and cover pages for papers.

This memo includes, as an appendix, the text of the ''Guide to Preparing Documents with –ms'' which contains additional examples of features of –ms.

This manual is a revision of, and replaces, ''Typing Documents on UNIX,'' dated November 22, 1974.

November 13, 1978

Typing Documents on the UNIX System: Using the –ms Macros with Troff and Nroff

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction. This memorandum describes a package of commands to produce papers using the *troff* and *nroff* formatting programs on the UNIX system. As with other *roff*-derived programs, text is prepared interspersed with formatting commands. However, this package, which itself is written in *troff* commands, provides higher-level commands than those provided with the basic *troff* program. The commands available in this package are listed in Appendix A.

Text. Type normally, except that instead of indenting for paragraphs, place a line reading ‘‘.PP’’ before each paragraph. This will produce indenting and extra space.

Alternatively, the command .LP that was used here will produce a left-aligned (block) paragraph. The paragraph spacing can be changed: see below under ‘‘Registers.’’

Beginning. For a document with a paper-type cover sheet, the input should start as follows:

```
[optional overall format .RP – see below]
.TL
Title of document (one or more lines)
.AU
Author(s) (may also be several lines)
.AI
Author's institution(s)
.AB
Abstract; to be placed on the cover sheet of a paper.
Line length is 5/6 of normal; use .ll here to change.
.AE  (abstract end)
text ... (begins with .PP, which see)
```

To omit some of the standard headings (e.g. no abstract, or no author’s institution) just omit the corresponding fields and command lines. The word ABSTRACT can be suppressed by writing ‘‘.AB no’’ for ‘‘.AB’’. Several interspersed .AU and .AI lines can be used for multiple authors. The headings are not compulsory: beginning with a .PP command is perfectly OK and will just start printing an ordinary paragraph. ***Warning:*** You can’t just begin a document with a line of text. Some –ms command must precede any text input. When in doubt, use .LP to get proper initialization, although any of the commands .PP, .LP, .TL, .SH, .NH is good enough. Figure 1 shows the legal arrangement of commands at the start of a document.

Cover Sheets and First Pages. The first line of a document signals the general format of the first page. In particular, if it is ".RP" a cover sheet with title and abstract is prepared. The default format is useful for scanning drafts.

In general –ms is arranged so that only one form of a document need be stored, containing all information; the first command gives the format, and unnecessary items for that format are ignored.

Warning: don’t put extraneous material between the .TL and .AE commands. Processing of the titling items is special, and other data placed in them may not behave as you expect. Don’t forget that some –ms command must precede any input text.

Page headings. The –ms macros, by default, will print a page heading containing a page number (if greater than 1). A default page footer is provided only in ***nroff***, where the date is used. The user can make minor adjustments to the page headings/footings by redefining the strings LH, CH, and RH which are the left, center and right portions of the page headings, respectively; and the strings LF, CF, and RF, which are the left, center and right portions of the page footer. For more complex formats, the user can redefine the macros PT and BT, which are invoked respectively at the top and bottom of each page. The margins (taken from registers HM and FM for the top and bottom margin respectively) are normally 1 inch; the page header/footer are in the middle of that space. The user who redefines these macros should be careful not to change parameters such as point size or font without resetting them to default values.

Multi-column formats. If you place the command ‘‘.2C’’ in your document, the document will be printed in double column format beginning at that point. This feature is not too useful in computer terminal output, but is often desirable on the typesetter. The command ‘‘.1C’’ will go back to one-column format and also skip to a new page. The ‘‘.2C’’ command is actually a special case of the command

```
.MC [column width [gutter width]]
```

which makes multiple columns with the specified column and gutter width; as many columns as will fit across the page are used. Thus triple, quadruple, ... column pages can be printed. Whenever the number of columns is changed (except going from full width to some larger number of columns) a new page is started.

Headings. To produce a special heading, there are two commands. If you type

```
.NH
type section heading here
may be several lines
```

you will get automatically numbered section headings (1, 2, 3, ...), in boldface. For example,

```
.NH
Care and Feeding of Department Heads
```

produces

1. Care and Feeding of Department Heads

Alternatively,

```
.SH
Care and Feeding of Directors
```

will print the heading with no number added:

Care and Feeding of Directors

Every section heading, of either type, should be followed by a paragraph beginning with .PP or .LP, indicating the end of the heading. Headings may contain more than one line of text.

The .NH command also supports more complex numbering schemes. If a numerical argument is given, it is taken to be a ‘‘level’’ number and an appropriate sub-section number is generated. Larger level numbers indicate deeper sub-sections, as in this example:

```
.NH
Erie-Lackawanna
.NH 2
Morris and Essex Division
.NH 3
Gladstone Branch
.NH 3
Montclair Branch
.NH 2
Boonton Line
```

generates:

2. Erie-Lackawanna

2.1. Morris and Essex Division

2.1.1. Gladstone Branch

2.1.2. Montclair Branch

2.2. Boonton Line

An explicit ‘‘.NH 0’’ will reset the numbering of level 1 to one, as here:

```
.NH 0
Penn Central
```

1. Penn Central

Indented paragraphs. (Paragraphs with hanging numbers, e.g. references.) The sequence

```
.IP [1]
Text for first paragraph, typed
normally for as long as you would
like on as many lines as needed.
.IP [2]
Text for second paragraph, ...
```

produces

[1] Text for first paragraph, typed normally for as long as you would like on as many lines as needed.

[2] Text for second paragraph, ...

A series of indented paragraphs may be followed by an ordinary paragraph beginning with .PP or .LP, depending on whether you wish indenting or not. The command .LP was used here.

More sophisticated uses of .IP are also possible. If the label is omitted, for example, a plain block indent is produced.

```
.IP
This material will
just be turned into a
block indent suitable for quotations or
such matter.
.LP
```

will produce

> This material will just be turned into a block indent suitable for quotations or such matter.

If a non-standard amount of indenting is required, it may be specified after the label (in character positions) and will remain in effect until the next .PP or .LP. Thus, the general form of the .IP command contains two additional fields: the label and the indenting length. For example,

```
.IP first: 9
Notice the longer label, requiring larger
indenting for these paragraphs.
.IP second:
And so forth.
.LP
```

produces this:

first: Notice the longer label, requiring larger indenting for these paragraphs.

second: And so forth.

It is also possible to produce multiple nested indents; the command .RS indicates that the next .IP starts from the current indentation level. Each .RE will eat up one level of indenting so you should balance .RS and .RE commands. The .RS command should be thought of as ‘‘move right’’ and the .RE command as ‘‘move left’’. As an example

```
.IP 1.
Bell Laboratories
.RS
.IP 1.1
Murray Hill
.IP 1.2
Holmdel
.IP 1.3
Whippany
.RS
.IP 1.3.1
Madison
.RE
.IP 1.4
Chester
.RE
.LP
```

will result in

1. Bell Laboratories
    - 1.1 Murray Hill
    - 1.2 Holmdel
    - 1.3 Whippany
        - 1.3.1 Madison
    - 1.4 Chester

All of these variations on .LP leave the right margin untouched. Sometimes, for purposes such as setting off a quotation, a paragraph indented on both right and left is required.

> A single paragraph like this is obtained by preceding it with .QP. More complicated material (several paragraphs) should be bracketed with .QS and .QE.

Emphasis. To get italics (on the typesetter) or underlining (on the terminal) say

```
.I
as much text as you want
can be typed here
.R
```

as was done for *these three words.* The .R command restores the normal (usually Roman) font. If only one word is to be italicized, it may be just given on the line with the .I command,

```
.I word
```

and in this case no .R is needed to restore the previous font. **Boldface** can be produced by

.B
Text to be set in boldface
goes here
.R

and also will be underlined on the terminal or line printer. As with .I, a single word can be placed in boldface by placing it on the same line as the .B command.

A few size changes can be specified similarly with the commands .LG (make larger), .SM (make smaller), and .NL (return to normal size). The size change is two points; the commands may be repeated for increased effect (here one .NL canceled two .SM commands).

If actual underlining as opposed to italicizing is required on the typesetter, the command

.UL word

will underline a word. There is no way to underline multiple words on the typesetter.

Footnotes. Material placed between lines with the commands .FS (footnote) and .FE (footnote end) will be collected, remembered, and finally placed at the bottom of the current page*. By default, footnotes are 11/12th the length of normal text, but this can be changed using the FL register (see below).

Displays and Tables. To prepare displays of lines, such as tables, in which the lines should not be re-arranged, enclose them in the commands .DS and .DE

.DS
table lines, like the
examples here, are placed
between .DS and .DE
.DE

By default, lines between .DS and .DE are indented and left-adjusted. You can also center lines, or retain the left margin. Lines bracketed by .DS C and .DE commands are centered (and not re-arranged); lines bracketed by .DS L and .DE are left-adjusted, not indented, and not re-arranged. A plain .DS is equivalent to .DS I, which indents and left-adjusts. Thus,

these lines were preceded
by .DS C and followed by
a .DE command;

whereas

these lines were preceded
by .DS L and followed by
a .DE command.

Note that .DS C centers each line; there is a variant .DS B that makes the display into a left-adjusted block of text, and then centers that entire block. Normally a display is kept together, on one page. If you wish to have a long display which may be split across page boundaries, use .CD, .LD, or .ID in place of the commands .DS C, .DS L, or .DS I respectively. An extra argument to the .DS I or .DS command is taken as an amount to indent. Note: it is tempting to assume that .DS R will right adjust lines, but it doesn't work.

Boxing words or lines. To draw rectangular boxes around words the command

.BX word

will print word as shown. The boxes will not be neat on a terminal, and this should not be used as a substitute for italics.

Longer pieces of text may be boxed by enclosing them with .B1 and .B2:

.B1
text...
.B2

as has been done here.

Keeping blocks together. If you wish to keep a table or other block of lines together on a page, there are ''keep - release'' commands. If a block of lines preceded by .KS and followed by .KE does not fit on the remainder of the current page, it will begin on a new page. Lines bracketed by .DS and .DE commands are automatically kept together this way. There is also a ''keep floating'' command: if the block to be kept together is preceded by .KF instead of .KS and does not fit on the current page, it will be moved down through the text until the top of the next page. Thus, no large blank space will be introduced in the document.

Nroff/Troff commands. Among the useful commands from the basic formatting programs are the following. They all work with both typesetter and computer terminal output:

* Like this.

.bp - begin new page.
.br - ‘‘break’’, stop running text
from line to line.
.sp n - insert n blank lines.
.na - don’t adjust right margins.

Date. By default, documents produced on computer terminals have the date at the bottom of each page; documents produced on the typesetter don’t. To force the date, say ‘‘.DA’’. To force no date, say ‘‘.ND’’. To lie about the date, say ‘‘.DA July 4, 1776’’ which puts the specified date at the bottom of each page. The command

.ND May 8, 1945

in ".RP" format places the specified date on the cover sheet and nowhere else. Place this line before the title.

Signature line. You can obtain a signature line by placing the command .SG in the document. The authors’ names will be output in place of the .SG line. An argument to .SG is used as a typing identification line, and placed after the signatures. The .SG command is ignored in released paper format.

Registers. Certain of the registers used by −ms can be altered to change default settings. They should be changed with .nr commands, as with

.nr PS 9

to make the default point size 9 point. If the effect is needed immediately, the normal *troff* command should be used in addition to changing the number register.

Register	Defines	Takes effect	Default
PS	point size	next para.	10
VS	line spacing	next para.	12 pts
LL	line length	next para.	6″
LT	title length	next para.	6″
PD	para. spacing	next para.	0.3 VS
PI	para. indent	next para.	5 ens
FL	footnote length	next FS	11/12 LL
CW	column width	next 2C	7/15 LL
GW	intercolumn gap	next 2C	1/15 LL
PO	page offset	next page	26/27″
HM	top margin	next page	1″
FM	bottom margin	next page	1″

You may also alter the strings LH, CH, and RH which are the left, center, and right headings respectively; and similarly LF, CF, and RF which are strings in the page footer. The page number on *output* is taken from register PN, to permit changing its output style. For more complicated headers and footers the macros PT and BT can be redefined, as explained earlier.

Accents. To simplify typing certain foreign words, strings representing common accent marks are defined. They precede the letter over which the mark is to appear. Here are the strings:

Input	Output	Input	Output
*′e	é	*˜a	ã
*`e	è	*Ce	ě
*:u	ü	*,c	ç
*^e	ê		

Use. After your document is prepared and stored on a file, you can print it on a terminal with the command*

nroff −ms file

and you can print it on the typesetter with the command

troff −ms file

(many options are possible). In each case, if your document is stored in several files, just list all the filenames where we have used ‘‘file’’. If equations or tables are used, *eqn* and/or *tbl* must be invoked as preprocessors.

References and further study. If you have to do Greek or mathematics, see ***eqn*** [1] for equation setting. To aid ***eqn*** users, ***−ms*** provides definitions of .EQ and .EN which normally center the equation and set it off slightly. An argument on .EQ is taken to be an equation number and placed in the right margin near the equation. In addition, there are three special arguments to EQ: the letters C, I, and L indicate centered (default), indented, and left adjusted equations, respectively. If there is both a format argument and an equation number, give the format argument first, as in

.EQ L (1.3a)

for a left-adjusted equation numbered (1.3a).

Similarly, the macros .TS and .TE are defined to separate tables (see [2]) from text with a little space. A very long table with a heading may be broken across pages by beginning it with .TS H instead of .TS, and placing the line .TH in the table data after the heading.

* If .2C was used, pipe the *nroff* output through *col;* make the first line of the input ‘‘.pi /usr/bin/col.’’

If the table has no heading repeated from page to page, just use the ordinary .TS and .TE macros.

To learn more about *troff* see [3] for a general introduction, and [4] for the full details (experts only). Information on related UNIX commands is in [5]. For jobs that do not seem well-adapted to −ms, consider other macro packages. It is often far easier to write a specific macro packages for such tasks as imitating particular journals than to try to adapt −ms.

Acknowledgment. Many thanks are due to Brian Kernighan for his help in the design and implementation of this package, and for his assistance in preparing this manual.

References

[1] B. W. Kernighan and L. L. Cherry, *Typesetting Mathematics — Users Guide (2nd edition),* Bell Laboratories Computing Science Report no. 17.

[2] M. E. Lesk, *Tbl — A Program to Format Tables,* Bell Laboratories Computing Science Report no. 45.

[3] B. W. Kernighan, *A Troff Tutorial,* Bell Laboratories, 1976.

[4] J. F. Ossanna, *Nroff/Troff Reference Manual,* Bell Laboratories Computing Science Report no. 51.

[5] K. Thompson and D. M. Ritchie, *UNIX Programmer's Manual,* Bell Laboratories, 1978.

Appendix A
List of Commands

1C	Return to single column format.
2C	Start double column format.
AB	Begin abstract.
AE	End abstract.
AI	Specify author’s institution.
AU	Specify author.
B	Begin boldface.
DA	Provide the date on each page.
DE	End display.
DS	Start display (also CD, LD, ID).
EN	End equation.
EQ	Begin equation.
FE	End footnote.
FS	Begin footnote.
I	Begin italics.
IP	Begin indented paragraph.
KE	Release keep.
KF	Begin floating keep.
KS	Start keep.
LG	Increase type size.
LP	Left aligned block paragraph.
ND	Change or cancel date.
NH	Specify numbered heading.
NL	Return to normal type size.
PP	Begin paragraph.
R	Return to regular font (usually Roman).
RE	End one level of relative indenting.
RP	Use released paper format.
RS	Relative indent increased one level.
SG	Insert signature line.
SH	Specify section heading.
SM	Change to smaller type size.
TL	Specify title.
UL	Underline one word.

Register Names

The following register names are used by –ms internally. Independent use of these names in one’s own macros may produce incorrect output. Note that no lower case letters are used in any –ms internal name.

Number registers used in –ms

:	DW	GW	HM	IQ	LL	NA	OJ	PO	T.	TV
#T	EF	H1	HT	IR	LT	NC	PD	PQ	TB	VS
1T	FL	H3	IK	KI	MM	NF	PF	PX	TD	YE
AV	FM	H4	IM	L1	MN	NS	PI	RO	TN	YY
CW	FP	H5	IP	LE	MO	OI	PN	ST	TQ	ZN

String registers used in –ms

′	A5	CB	DW	EZ	I	KF	MR	R1	RT	TL
`	AB	CC	DY	FA	I1	KQ	ND	R2	S0	TM
ˆ	AE	CD	E1	FE	I2	KS	NH	R3	S1	TQ
˜	AI	CF	E2	FJ	I3	LB	NL	R4	S2	TS
:	AU	CH	E3	FK	I4	LD	NP	R5	SG	TT
,	B	CM	E4	FN	I5	LG	OD	RC	SH	UL
1C	BG	CS	E5	FO	ID	LP	OK	RE	SM	WB
2C	BT	CT	EE	FQ	IE	ME	PP	RF	SN	WH
A1	C	D	EL	FS	IM	MF	PT	RH	SY	WT
A2	C1	DA	EM	FV	IP	MH	PY	RP	TA	XD
A3	C2	DE	EN	FY	IZ	MN	QF	RQ	TE	XF
A4	CA	DS	EQ	HO	KE	MO	R	RS	TH	XK

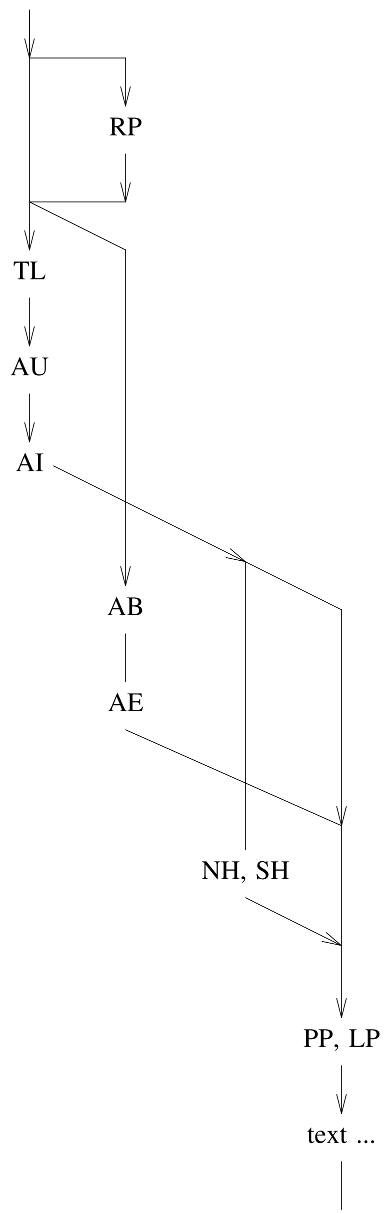


Figure 1

A Guide to Preparing Documents with −ms

M. E. Lesk

Bell Laboratories August 1978

This guide gives some simple examples of document preparation on Bell Labs computers, emphasizing the use of the *−ms* macro package. It enormously abbreviates information in
1. *Typing Documents on UNIX and GCOS,* by M. E. Lesk;
2. *Typesetting Mathematics – User's Guide,* by B. W. Kernighan and L. L. Cherry; and
3. *Tbl – A Program to Format Tables,* by M. E. Lesk.

These memos are all included in the *UNIX Programmer's Manual, Volume 2.* The new user should also have *A Tutorial Introduction to the UNIX Text Editor,* by B. W. Kernighan.

For more detailed information, read *Advanced Editing on UNIX* and *A Troff Tutorial,* by B. W. Kernighan, and (for experts) *Nroff/Troff Reference Manual* by J. F. Ossanna. Information on related commands is found (for UNIX users) in *UNIX for Beginners* by B. W. Kernighan and the *UNIX Programmer's Manual* by K. Thompson and D. M. Ritchie.

Contents

Throughout the examples, input is shown in
this Helvetica sans serif font
while the resulting output is shown in
this Times Roman font.

UNIX Document no. 1111

Commands for a TM

```
.TM 1978-5b3 99999 99999-11
.ND April 1, 1976
.TL
The Role of the Allen Wrench in Modern
Electronics
.AU "MH 2G-111" 2345
J. Q. Pencilpusher
.AU "MH 1K-222" 5432
X. Y. Hardwired
.AI
.MH
.OK
Tools
Design
.AB
This abstract should be short enough to
fit on a single page cover sheet.
It must attract the reader into sending for
the complete memorandum.
.AE
.CS 10 2 12 5 6 7
.NH
Introduction.
.PP
Now the first paragraph of actual text ...
...
Last line of text.
.SG MH-1234-JQP/XYH-unix
.NH
References ...
```

Commands not needed in a particular format are ignored.

Bell Laboratories Cover Sheet for TM

This information is for employees of Bell Laboratories. (GEI 13.9-3)

Title- The Role of the Allen Wrench in Modern Electronics Date- April 1, 1976

TM- 1978-5b3

Other Keywords- Tools
Design

Author	Location	Ext.	
J. Q. Pencilpusher	MH 2G-111	2345	Charging Case- **99999**
X. Y. Hardwired	MH 1K-222	5432	Filing Case- **99999a**

ABSTRACT

This abstract should be short enough to fit on a single page cover sheet. It must attract the reader into sending for the complete memorandum.

Pages Text 10	Other 2	Total 12
No. Figures 5	No. Tables 6	No. Refs. 7

E-1932-U (6-73) SEE REVERSE SIDE FOR DISTRIBUTION LIST

A Released Paper with Mathematics

```
.EQ
delim $$
.EN
.RP
```

... (as for a TM)

```
.CS 10 2 12 5 6 7
.NH
Introduction
.PP
The solution to the torque handle equation
.EQ (1)
sum from 0 to inf F ( x sub i ) = G ( x )
.EN
is found with the transformation $ x = rho over
theta $ where $ rho = G prime (x) $ and $theta$
is derived ...
```

The Role of the Allen Wrench
in Modern Electronics

J. Q. Pencilpusher

X. Y. Hardwired

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This abstract should be short enough to fit on a single page cover sheet. It must attract the reader into sending for the complete memorandum.

April 1, 1976

The Role of the Allen Wrench
in Modern Electronics

J. Q. Pencilpusher

X. Y. Hardwired

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

The solution to the torque handle equation

$$\sum_0^\infty F(x_i) = G(x) \qquad (1)$$

is found with the transformation $x = \frac{\rho}{\theta}$ where $\rho = G'(x)$ and θ is derived from well-known principles.

An Internal Memorandum

```
.IM
.ND January 24, 1956
.TL
The 1956 Consent Decree
.AU
Able, Baker &
Charley, Attys.
.PP
Plaintiff, United States of America, having filed
its complaint herein on January 14, 1949; the
defendants having appeared and filed their
answer to such complaint denying the
substantive allegations thereof; and the parties,
by their attorneys, ...
```

Bell Laboratories

Subject: The 1956 Consent Decree | date: January 24, 1956

from: Able, Baker &
Charley, Attys.

Plaintiff, United States of America, having filed its complaint herein on January 14, 1949; the defendants having appeared and filed their answer to such complaint denying the substantive allegations thereof; and the parties, by their attorneys, having severally consented to the entry of this Final Judgment without trial or adjudication of any issues of fact or law herein and without this Final Judgment constituting any evidence or admission by any party in respect of any such issues;

Now, therefore before any testimony has been taken herein, and without trial or adjudication of any issue of fact or law herein, and upon the consent of all parties hereto, it is hereby

Ordered, adjudged and decreed as follows:

I. [Sherman Act]

This Court has jurisdiction of the subject matter herein and of all the parties hereto. The complaint states a claim upon which relief may be granted against each of the defendants under Sections 1, 2 and 3 of the Act of Congress of July 2, 1890, entitled ‘‘An act to protect trade and commerce against unlawful restraints and monopolies,’’ commonly known as the Sherman Act, as amended.

II. [Definitions]

For the purposes of this Final Judgment:

(a) ‘‘Western’’ shall mean the defendant Western Electric Company, Incorporated.

Other formats possible (specify before .TL) are: .MR (‘‘memo for record’’), .MF (‘‘memo for file’’), .EG (‘‘engineer’s notes’’) and .TR (Computing Science Tech. Report).

Headings

```
.NH
Introduction.
.PP
text text text
```

```
.SH
Appendix I
.PP
text text text
```

1. Introduction

text text text

Appendix I

text text text

A Simple List

```
.IP 1.
J. Pencilpusher and X. Hardwired,
.I
A New Kind of Set Screw,
.R
Proc. IEEE
.B 75
(1976), 23-255.
.IP 2.
H. Nails and R. Irons,
.I
Fasteners for Printed Circuit Boards,
.R
Proc. ASME
.B 23
(1974), 23-24.
.LP
```
(terminates list)

1. J. Pencilpusher and X. Hardwired, *A New Kind of Set Screw,* Proc. IEEE **75** (1976), 23-255.
2. H. Nails and R. Irons, *Fasteners for Printed Circuit Boards,* Proc. ASME **23** (1974), 23-24.

Displays

```
text text text text text text
.DS
and now
for something
completely different
.DE
text text text text text text
```

hoboken harrison newark roseville avenue grove street east orange brick church orange highland avenue mountain station south orange maplewood millburn short hills summit new providence

> and now
> for something
> completely different

murray hill berkeley heights gillette stirling millington lyons basking ridge bernardsville far hills peapack gladstone

Options: `.DS L`: left-adjust; `.DS C`: line-by-line center; `.DS B`: make block, then center.

Footnotes

```
Among the most important occupants
of the workbench are the long-nosed pliers.
Without these basic tools*
.FS
* As first shown by Tiger & Leopard
(1975).
.FE
few assemblies could be completed.  They may
lack the popular appeal of the sledgehammer
```

Among the most important occupants of the workbench are the long-nosed pliers. Without these basic tools* few assemblies could be completed. They may lack the popular appeal of the sledgehammer

* As first shown by Tiger & Leopard (1975).

Multiple Indents

```
This is ordinary text to point out
the margins of the page.
.IP 1.
First level item
.RS
.IP a)
Second level.
.IP b)
Continued here with another second
level item, but somewhat longer.
.RE
.IP 2.
Return to previous value of the
indenting at this point.
.IP 3.
Another
line.
```

This is ordinary text to point out the margins of the page.

1. First level item
   a) Second level.
   b) Continued here with another second level item, but somewhat longer.
2. Return to previous value of the indenting at this point.
3. Another line.

Keeps

Lines bracketed by the following commands are kept together, and will appear entirely on one page:

`.KS`	not moved	`.KF`	may float
`.KE`	through text	`.KE`	in text

Double Column

```
.TL
The Declaration of Independence
.2C
.PP
When in the course of human events, it becomes
necessary for one people to dissolve the political
bonds which have connected them with another, and
to assume among the powers of the earth the
separate and equal station to which the laws of
Nature and of Nature's God entitle them, a decent
respect to the opinions of
```

The Declaration of Independence

When in the course of human events, it becomes necessary for one people to dissolve the political bonds which have connected them with another, and to assume among the powers of the earth the separate and equal station to which the laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

We hold these truths to be self-evident, that all men are created equal, that they are endowed by their creator with certain unalienable rights, that among these are life, liberty, and the pursuit of happiness. That to secure these rights, governments are instituted among men,

Equations

```
A displayed equation is marked
with an equation number at the right margin
by adding an argument to the EQ line:
.EQ (1.3)
x sup 2 over a sup 2 ~=~ sqrt {p z sup 2 +qz+r}
.EN
```

A displayed equation is marked with an equation number at the right margin by adding an argument to the EQ line:

$$\frac{x^2}{a^2} = \sqrt{pz^2+qz+r} \qquad (1.3)$$

```
.EQ  I (2.2a)
bold V bar sub nu~=~left [ pile {a above b above
c } right ] + left [ matrix { col { A(11) above .
above . } col { . above . above .} col {. above .
above A(33) }} right ] cdot left [ pile { alpha
above beta above  gamma } right ]
.EN
```

$$\bar{\mathbf{V}}_\nu = \begin{bmatrix} a \\ b \\ c \end{bmatrix} + \begin{bmatrix} A(11) & . & . \\ . & . & . \\ . & . & A(33) \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \qquad (2.2a)$$

```
.EQ  L
F hat ( chi ) ~ mark = ~ | del V | sup 2
.EN
.EQ  L
lineup =~ {left ( {partial V} over {partial x} right ) }
sup 2 + { left ( {partial V} over {partial y} right ) }
sup 2 ~~~~~~ lambda -> inf
.EN
```

$$\hat{F}(\chi) = |\nabla V|^2$$

$$= \left[\frac{\partial V}{\partial x}\right]^2 + \left[\frac{\partial V}{\partial y}\right]^2 \qquad \lambda \to \infty$$

```
$ a dot $,  $ b dotdot$,  $ xi tilde times y vec$:
```

$\dot{a}$, $\ddot{b}$, $\tilde{\xi} \times \vec{y}$. (with delim $$ on, see panel 3).

See also the equations in the second table, panel 8.

Some Registers You Can Change

Line length
  .nr LL 7i

Title length
  .nr LT 7i

Point size
  .nr PS 9

Vertical spacing
  .nr VS 11

Column width
  .nr CW 3i

Intercolumn spacing
  .nr GW .5i

Margins – head and foot
  .nr HM .75i
  .nr FM .75i

Paragraph indent
  .nr PI 2n

Paragraph spacing
  .nr PD 0

Page offset
  .nr PO 0.5i

Page heading
  .ds CH Appendix
    (center)
  .ds RH 7-25-76
    (right)
  .ds LH Private
    (left)

Page footer
  .ds CF Draft
  .ds LF
  .ds RF   similar

Page numbers
  .nr % 3

Tables

.TS (ⓣ indicates a tab)

```
allbox;
c s s
c c c
n n n.
AT&T Common Stock
YearⓣPriceⓣDividend
1971ⓣ41-54ⓣ$2.60
2ⓣ41-54ⓣ2.70
3ⓣ46-55ⓣ2.87
4ⓣ40-53ⓣ3.24
5ⓣ45-52ⓣ3.40
6ⓣ51-59ⓣ.95*
.TE
* (first quarter only)
```

AT&T Common Stock		
Year	Price	Dividend
1971	41-54	$2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

The meanings of the key-letters describing the alignment of each entry are:

c	center	n	numerical
r	right-adjust	a	subcolumn
l	left-adjust	s	spanned

The global table options are center, expand, box, doublebox, allbox, tab (x) and linesize (n).

.TS (with delim $$ on, see panel 3)

```
doublebox, center;
c c
l l.
NameⓣDefinition
.sp
Gammaⓣ$GAMMA (z) = int sub 0 sup inf \
   t sup {z-1} e sup -t dt$
Sineⓣ$sin (x) = 1 over 2i ( e sup ix - e sup -ix )$
Errorⓣ$ roman erf (z) = 2 over sqrt pi \
  int sub 0 sup z e sup {-t sup 2} dt$
Besselⓣ$ J sub 0 (z) = 1 over pi \
  int sub 0 sup pi cos ( z sin theta ) d theta $
Zetaⓣ$ zeta (s) = \
  sum from k=1 to inf k sup -s ~~( Re~s > 1)$
.TE
```

Name	Definition
Gamma	$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$
Sine	$\sin(x) = \frac{1}{2i}(e^{ix} - e^{-ix})$
Error	$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$
Bessel	$J_0(z) = \frac{1}{\pi} \int_0^\pi \cos(z \sin\theta) d\theta$
Zeta	$\zeta(s) = \sum_{k=1}^\infty k^{-s} \quad (\mathrm{Re}\ s > 1)$

Usage

Documents with just text:
troff -ms files
With equations only:
eqn files | troff -ms
With tables only:
tbl files | troff -ms
With both tables and equations:
tbl files | eqn | troff -ms

The above generates STARE output on GCOS: replace –st with –ph for typesetter output.

A System for Typesetting Mathematics

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper describes the design and implementation of a system for typesetting mathematics. The language has been designed to be easy to learn and to use by people (for example, secretaries and mathematical typists) who know neither mathematics nor typesetting. Experience indicates that the language can be learned in an hour or so, for it has few rules and fewer exceptions. For typical expressions, the size and font changes, positioning, line drawing, and the like necessary to print according to mathematical conventions are all done automatically. For example, the input

sum from i=0 to infinity x sub i = pi over 2

produces

$$\sum_{i=0}^{\infty} x_i = \frac{\pi}{2}$$

The syntax of the language is specified by a small context-free grammar; a compiler-compiler is used to make a compiler that translates this language into typesetting commands. Output may be produced on either a phototypesetter or on a terminal with forward and reverse half-line motions. The system interfaces directly with text formatting programs, so mixtures of text and mathematics may be handled simply.

This paper is a revision of a paper originally published in CACM, March, 1975.

1. Introduction

‘‘Mathematics is known in the trade as *difficult,* or *penalty, copy* because it is slower, more difficult, and more expensive to set in type than any other kind of copy normally occurring in books and journals.’’ [1]

One difficulty with mathematical text is the multiplicity of characters, sizes, and fonts. An expression such as

$$\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$$

requires an intimate mixture of roman, italic and greek letters, in three sizes, and a special character or two. (‘‘Requires’’ is perhaps the wrong word, but mathematics has its own typographical conventions which are quite different from those of ordinary text.) Typesetting such an expression by traditional methods is still an essentially manual operation.

A second difficulty is the two dimensional character of mathematics, which the superscript and limits in the preceding example showed in its simplest form. This is carried further by

$$a_0 + \cfrac{b_1}{a_1 + \cfrac{b_2}{a_2 + \cfrac{b_3}{a_3 + \cdots}}}$$

and still further by

$$\int \frac{dx}{ae^{mx} - be^{-mx}} = \begin{cases} \dfrac{1}{2m\sqrt{ab}} \log \dfrac{\sqrt{a}\,e^{mx} - \sqrt{b}}{\sqrt{a}\,e^{mx} + \sqrt{b}} \\ \dfrac{1}{m\sqrt{ab}} \tanh^{-1}\left(\dfrac{\sqrt{a}}{\sqrt{b}} e^{mx}\right) \\ \dfrac{-1}{m\sqrt{ab}} \coth^{-1}\left(\dfrac{\sqrt{a}}{\sqrt{b}} e^{mx}\right) \end{cases}$$

These examples also show line-drawing, built-up characters like braces and radicals, and a spectrum of positioning problems. (Section 6 shows what a user has to type to produce these on our system.)

2. Photocomposition

Photocomposition techniques can be used to solve some of the problems of typesetting mathematics. A phototypesetter is a device which exposes a piece of photographic paper or film, placing characters wherever they are wanted. The Graphic Systems phototypesetter[2] on the UNIX operating system[3] works by shining light through a character stencil. The character is made the right size by lenses, and the light beam directed by fiber optics to the desired place on a piece of photographic paper. The exposed paper is developed and typically used in some form of photo-offset reproduction.

On UNIX, the phototypesetter is driven by a formatting program called TROFF [4]. TROFF was designed for setting running text. It also provides all of the facilities that one needs for doing mathematics, such as arbitrary horizontal and vertical motions, line-drawing, size changing, but the syntax for describing these special operations is difficult to learn, and difficult even for experienced users to type correctly.

For this reason we decided to use TROFF as an ''assembly language,'' by designing a language for describing mathematical expressions, and compiling it into TROFF.

3. Language Design

The fundamental principle upon which we based our language design is that the language should be easy to use by people (for example, secretaries) who know neither mathematics nor typesetting.

This principle implies several things. First, ''normal'' mathematical conventions about operator precedence, parentheses, and the like cannot be used, for to give special meaning to such characters means that the user has to understand what he or she is typing. Thus the language should not assume, for instance, that parentheses are always balanced, for they are not in the half-open interval $(a,b]$. Nor should it assume that that $\sqrt{a+b}$ can be replaced by $(a+b)^{1/2}$, or that $1/(1-x)$ is better written as $\frac{1}{1-x}$ (or vice versa).

Second, there should be relatively few rules, keywords, special symbols and operators, and the like. This keeps the language easy to learn and remember. Furthermore, there should be few exceptions to the rules that do exist: if something works in one situation, it should work everywhere. If a variable can have a subscript, then a subscript can have a subscript, and so on without limit.

Third, ''standard'' things should happen automatically. Someone who types ''x=y+z+1'' should get ''$x=y+z+1$''. Subscripts and superscripts should automatically be printed in an appropriately smaller size, with no special intervention. Fraction bars have to be made the right length and positioned at the right height. And so on. Indeed a mechanism for overriding default actions has to exist, but its application is the exception, not the rule.

We assume that the typist has a reasonable picture (a two-dimensional representation) of the desired final form, as might be handwritten by the author of a paper. We also assume that the input is typed on a computer terminal much like an ordinary typewriter. This implies an input alphabet of perhaps 100 characters, none of them special.

A secondary, but still important, goal in our design was that the system should be easy to implement, since neither of the authors had any desire to make a long-term project of it. Since our design was not firm, it was also necessary that the program be easy to change at any time.

To make the program easy to build and to change, and to guarantee regularity (''it should work everywhere''), the language is defined by a context-free grammar, described in Section 5. The compiler for the language was built using a compiler-compiler.

A priori, the grammar/compiler-compiler approach seemed the right thing to do. Our subsequent experience leads us to believe that any other course would have been folly. The original language was designed in a few days. Construction of a working system sufficient to try significant examples required perhaps a person-month. Since then, we have spent a modest amount of additional time over several years tuning, adding facilities, and occasionally changing the language as users make criticisms and suggestions.

We also decided quite early that we would let TROFF do our work for us whenever possible. TROFF is quite a powerful program, with a macro facility, text and arithmetic variables, numerical computation and testing, and conditional branching. Thus we have been able to avoid writing a lot of mundane but tricky software. For example, we store no text strings, but simply pass them on to TROFF. Thus we avoid having to write a storage management package. Furthermore, we have been able to isolate ourselves from most details of the particular device and character set currently in use. For example, we let TROFF compute the widths of all strings of characters; we need know nothing about them.

A third design goal is special to our environment. Since our program is only useful for typesetting mathematics, it is necessary that it interface cleanly with the underlying typesetting language for the benefit of users who want to set intermingled mathematics and text (the usual case). The standard mode of operation is that when a document is typed, mathematical expressions are input as part of the text, but marked by user settable delimiters. The program reads this input and treats as comments those things which are not mathematics, simply passing them through untouched. At the same time it converts the mathematical input into the necessary TROFF commands. The resulting ioutput is passed directly to TROFF where the comments and the mathematical parts both become text and/or TROFF commands.

4. The Language

We will not try to describe the language precisely here; interested readers may refer to the appendix for more details. Throughout this section, we will write expressions exactly as they are handed to the typesetting program (hereinafter called ''EQN''), except that we won't show the delimiters that the user types to mark the beginning and end of the expression. The interface between EQN and TROFF is described at the end of this section.

As we said, typing x=y+z+1 should produce $x=y+z+1$, and indeed it does. Variables are made italic, operators and digits become roman, and normal spacings between letters and operators are altered slightly to give a more pleasing appearance.

Input is free-form. Spaces and new lines in the input are used by EQN to separate pieces of the input; they are not used to create space in the output. Thus

```
x    =    y
   + z + 1
```

also gives $x=y+z+1$. Free-form input is easier to type initially; subsequent editing is also easier, for an expression may be typed as many short lines.

Extra white space can be forced into the output by several characters of various sizes. A tilde ''~'' gives a space equal to the normal word spacing in text; a circumflex gives half this much, and a tab charcter spaces to the next tab stop.

Spaces (or tildes, etc.) also serve to delimit pieces of the input. For example, to get

$$f(t)=2\pi\int \sin(\omega t)dt$$

we write

```
f(t) = 2 pi int sin ( omega t )dt
```

Here spaces are *necessary* in the input to indicate that *sin, pi, int,* and *omega* are special, and potentially worth special treatment. EQN looks up each such string of characters in a table, and if appropriate gives it a translation. In this case, *pi* and *omega* become their greek equivalents, *int* becomes the integral sign (which must be moved down and enlarged so it looks ''right''), and *sin* is made roman, following conventional mathematical practice. Parentheses, digits and operators are automatically made roman wherever found.

Fractions are specified with the keyword *over:*

```
a+b over c+d+e = 1
```

produces

$$\frac{a+b}{c+d+e}=1$$

Similarly, subscripts and superscripts are introduced by the keywords *sub* and *sup:*

$$x^2+y^2=z^2$$

is produced by

```
x sup 2 + y sup 2 = z sup 2
```

The spaces after the 2's are necessary to mark the end of the superscripts; similarly the keyword *sup* has to be marked off by spaces or some equivalent delimiter. The return to the proper baseline is automatic. Multiple levels of subscripts or superscripts are of course allowed: ''x sup y sup z'' is x^{y^z}. The construct ''something *sub* something *sup* something'' is recognized as a special case, so ''x sub i sup 2'' is x_i^2 instead of ${x_i}^2$.

More complicated expressions can now be formed with these primitives:

$$\frac{\partial^2 f}{\partial x^2}=\frac{x^2}{a^2}+\frac{y^2}{b^2}$$

is produced by

```
{partial sup 2 f} over {partial x sup 2} =
x sup 2 over a sup 2 + y sup 2 over b sup 2
```

Braces {} are used to group objects together; in this case they indicate unambiguously what goes over what on the left-hand side of the expression. The language defines the precedence of *sup* to be higher than that of *over,* so no braces are needed to get the correct association on the right side. Braces can always be used when in doubt about precedence.

The braces convention is an example of the power of using a recursive grammar to define the language. It is part of the language that if a construct can appear in some context, then *any expression* in braces can also occur in that context.

There is a *sqrt* operator for making square roots of the appropriate size: ''sqrt a+b'' produces $\sqrt{a+b}$, and

```
x = {–b +– sqrt{b sup 2 –4ac}} over 2a
```

is

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

Since large radicals look poor on our typesetter, *sqrt* is not useful for tall expressions.

Limits on summations, integrals and similar constructions are specified with the keywords *from* and *to.* To get

$$\sum_{i=0}^{\infty} x_i \rightarrow 0$$

we need only type

sum from i=0 to inf x sub i –> 0

Centering and making the Σ big enough and the limits smaller are all automatic. The *from* and *to* parts are both optional, and the central part (e.g., the Σ) can in fact be anything:

lim from {x –> pi /2} (tan~x) = inf

is

$$\lim_{x \to \pi/2} (\tan x) = \infty$$

Again, the braces indicate just what goes into the *from* part.

There is a facility for making braces, brackets, parentheses, and vertical bars of the right height, using the keywords *left* and *right:*

left [x+y over 2a right]~=~1

makes

$$\left[\frac{x+y}{2a} \right] = 1$$

A *left* need not have a corresponding *right,* as we shall see in the next example. Any characters may follow *left* and *right,* but generally only various parentheses and bars are meaningful.

Big brackets, etc., are often used with another facility, called *piles,* which make vertical piles of objects. For example, to get

$$sign(x) \equiv \left\{ \begin{array}{rll} 1 & \text{if} & x>0 \\ 0 & \text{if} & x=0 \\ -1 & \text{if} & x<0 \end{array} \right.$$

we can type

```
sign (x) ~==~ left {
   rpile {1 above 0 above –1}
   ~~lpile {if above if above if}
   ~~lpile {x>0 above x=0 above x<0}
```

The construction ‘‘left {’’ makes a left brace big enough to enclose the ‘‘rpile {...}’’, which is a right-justified pile of ‘‘above ... above ...’’. ‘‘lpile’’ makes a left-justified pile. There are also centered piles. Because of the recursive language definition, a pile can contain any number of elements; any element of a pile can of course contain piles.

Although EQN makes a valiant attempt to use the right sizes and fonts, there are times when the default assumptions are simply not what is wanted. For instance the italic *sign* in the previous example would conventionally be in roman. Slides and transparencies often require larger characters than normal text. Thus we also provide size and font changing commands: ‘‘size 12 bold {A~x~=~y}’’ will produce $\mathbf{A\ x = y}$. *Size* is followed by a number representing a character size in points. (One point is 1/72 inch; this paper is set in 9 point type.)

If necessary, an input string can be quoted in "...", which turns off grammatical significance, and any font or spacing changes that might otherwise be done on it. Thus we can say

lim~ roman "sup" ~x sub n = 0

to ensure that the supremum doesn't become a superscript:

$$\lim\ \mathrm{sup}\ x_n = 0$$

Diacritical marks, long a problem in traditional typesetting, are straightforward:

$$\underline{\dot{x}} + \hat{x} + \tilde{y} + \hat{X} + \ddot{Y} = \overline{z+Z}$$

is made by typing

```
x dot under + x hat + y tilde
+ X hat + Y dotdot = z+Z bar
```

There are also facilities for globally changing default sizes and fonts, for example for making viewgraphs or for setting chemical equations. The language allows for matrices, and for lining up equations at the same horizontal position.

Finally, there is a definition facility, so a user can say

define name "..."

at any time in the document; henceforth, any occurrence of the token ‘‘name’’ in an expression will be expanded into whatever was inside the double quotes in its definition. This lets users tailor the language to their own specifications, for it is quite possible to redefine keywords like *sup* or *over.* Section 6 shows an example of definitions.

The EQN preprocessor reads intermixed text and equations, and passes its output to TROFF. Since TROFF uses lines beginning with a period as control words (e.g., ‘‘.ce’’ means ‘‘center the next output line’’), EQN uses the sequence ‘‘.EQ’’ to mark the beginning of an equation and ‘‘.EN’’ to mark the end. The ‘‘.EQ’’ and ‘‘.EN’’ are passed through to TROFF untouched, so they can also be used by a knowledgeable user to center equations, number them automatically, etc. By default, however, ‘‘.EQ’’ and ‘‘.EN’’ are simply ignored by TROFF, so by default equations are printed in-line.

‘‘.EQ’’ and ‘‘.EN’’ can be supplemented by TROFF commands as desired; for example, a centered display equation can be produced with the input:

```
.ce
.EQ
x sub i = y sub i ...
.EN
```

Since it is tedious to type ‘‘.EQ’’ and ‘‘.EN’’ around very short expressions (single letters, for instance), the user can also define two characters to serve as the left and right delimiters of expressions. These characters are recognized anywhere in subsequent text. For example if the left and right delimiters have both been set to ‘‘#’’, the input:

Let #x sub i#, #y# and #alpha# be positive

produces:

Let x_i, y and α be positive

Running a preprocessor is strikingly easy on UNIX. To typeset text stored in file ‘‘f’’, one issues the command:

eqn f | troff

The vertical bar connects the output of one process (EQN) to the input of another (TROFF).

5. Language Theory

The basic structure of the language is not a particularly original one. Equations are pictured as a set of ‘‘boxes,’’ pieced together in various ways. For example, something with a subscript is just a box followed by another box moved downward and shrunk by an appropriate amount. A fraction is just a box centered above another box, at the right altitude, with a line of correct length drawn between them.

The grammar for the language is shown below. For purposes of exposition, we have collapsed some productions. In the original grammar, there are about 70 productions, but many of these are simple ones used only to guarantee that some keyword is recognized early enough in the parsing process. Symbols in capital letters are terminal symbols; lower case symbols are non-terminals, i.e., syntactic categories. The vertical bar | indicates an alternative; the brackets [ ] indicate optional material. A TEXT is a string of non-blank characters or any string inside double quotes; the other terminal symbols represent literal occurrences of the corresponding keyword.

```
eqn   : box | eqn box

box   : text
      | { eqn }
      | box OVER box
      | SQRT box
      | box SUB box | box SUP box
      | [ L | C | R ]PILE { list }
      | LEFT text eqn [ RIGHT text ]
      | box [ FROM box ] [ TO box ]
      | SIZE text box
      | [ROMAN | BOLD | ITALIC] box
      | box [HAT | BAR | DOT | DOTDOT | TILDE]
      | DEFINE text text

list  : eqn | list ABOVE eqn

text  : TEXT
```

The grammar makes it obvious why there are few exceptions. For example, the observation that something can be replaced by a more complicated something in braces is implicit in the productions:

```
eqn  : box | eqn box
box  : text | { eqn }
```

Anywhere a single character could be used, *any* legal construction can be used.

Clearly, our grammar is highly ambiguous. What, for instance, do we do with the input

a over b over c ?

Is it

{a over b} over c

or is it

a over {b over c} ?

To answer questions like this, the grammar is supplemented with a small set of rules that describe the precedence and associativity of operators. In particular, we specify (more or less arbitrarily) that *over* associates to the left, so the first alternative above is the one chosen. On the other hand, *sub* and *sup* bind to the right, because this is closer to standard mathematical practice. That is, we assume x^{a^b} is $x^{(a^b)}$, not $(x^a)^b$.

The precedence rules resolve the ambiguity in a construction like

a sup 2 over b

We define *sup* to have a higher precedence than *over,* so this construction is parsed as $\frac{a^2}{b}$ instead of $a^{\frac{2}{b}}$.

Naturally, a user can always force a particular parsing by placing braces around expressions.

The ambiguous grammar approach seems to be quite useful. The grammar we use is small enough to be easily understood, for it contains none of the productions that would be normally used for resolving ambiguity. Instead the supplemental information about precedence and associativity (also small enough to be understood) provides the compiler-compiler with the information it needs to make a fast, deterministic parser for the specific language we want. When the language is supplemented by the disambiguating rules, it is in fact LR(1) and thus easy to parse[5].

The output code is generated as the input is scanned. Any time a production of the grammar is recognized, (potentially) some TROFF commands are output. For example, when the lexical analyzer reports that it has found a TEXT (i.e., a string of contiguous characters), we have recognized the production:

text : TEXT

The translation of this is simple. We generate a local name for the string, then hand the name and the string to TROFF, and let TROFF perform the storage management. All we save is the name of the string, its height, and its baseline.

As another example, the translation associated with the production

```
box    : box OVER box
```

is:

Width of output box =
  slightly more than largest input width
Height of output box =
  slightly more than sum of input heights
Base of output box =
  slightly more than height of bottom input box
String describing output box =
  move down;
  move right enough to center bottom box;
  draw bottom box (i.e., copy string for bottom box);
  move up; move left enough to center top box;
  draw top box (i.e., copy string for top box);
  move down and left; draw line full width;
  return to proper base line.

Most of the other productions have equally simple semantic actions. Picturing the output as a set of properly placed boxes makes the right sequence of positioning commands quite obvious. The main difficulty is in finding the right numbers to use for esthetically pleasing positioning.

With a grammar, it is usually clear how to extend the language. For instance, one of our users suggested a TENSOR operator, to make constructions like

$${}^{l}_{m}\mathbf{T}^{kj}_{ni}$$

Grammatically, this is easy: it is sufficient to add a production like

```
box    : TENSOR { list }
```

Semantically, we need only juggle the boxes to the right places.

6. Experience

There are really three aspects of interest—how well EQN sets mathematics, how well it satisfies its goal of being ‘‘easy to use,’’ and how easy it was to build.

The first question is easily addressed. This entire paper has been set by the program. Readers can judge for themselves whether it is good enough for their purposes. One of our users commented that although the output is not as good as the best hand-set material, it is still better than average, and much better than the worst. In any case, who cares? Printed books cannot compete with the birds and flowers of illuminated manuscripts on esthetic grounds, either, but they have some clear economic advantages.

Some of the deficiencies in the output could be cleaned up with more work on our part. For example, we sometimes leave too much space between a roman letter and an italic one. If we were willing to keep track of the fonts involved, we could do this better more of the time.

Some other weaknesses are inherent in our output device. It is hard, for instance, to draw a line of an arbitrary length without getting a perceptible overstrike at one end.

As to ease of use, at the time of writing, the system has been used by two distinct groups. One user population consists of mathematicians, chemists, physicists, and computer scientists. Their typical reaction has been something like:

(1) It’s easy to write, although I make the following mistakes...

(2) How do I do...?

(3) It botches the following things.... Why don’t you fix them?

(4) You really need the following features...

The learning time is short. A few minutes gives the general flavor, and typing a page or two of a paper generally uncovers most of the misconceptions about how it works.

The second user group is much larger, the secretaries and mathematical typists who were the original target of the system. They tend to be enthusiastic converts. They find the language easy to learn (most are largely self-taught), and have little trouble producing the output they want. They are of course less critical of the esthetics of their output than users trained in mathematics. After a transition period, most find using a computer more interesting than a regular typewriter.

The main difficulty that users have seems to be remembering that a blank is a delimiter; even experienced users use blanks where they shouldn’t and omit them when they are needed. A common instance is typing

```
f(x sub i)
```

which produces

$$f\,(x_{i)}$$

instead of

$$f\,(x_i)$$

Since the EQN language knows no mathematics, it cannot deduce that the right parenthesis is not part of the subscript.

The language is somewhat prolix, but this doesn't seem excessive considering how much is being done, and it is certainly more compact than the corresponding TROFF commands. For example, here is the source for the continued fraction expression in Section 1 of this paper:

```
a sub 0 + b sub 1 over
  {a sub 1 + b sub 2 over
    {a sub 2 + b sub 3 over
      {a sub 3 + ... }}}
```

This is the input for the large integral of Section 1; notice the use of definitions:

```
define emx "{e sup mx}"
define mab "{m sqrt ab}"
define sa "{sqrt a}"
define sb "{sqrt b}"
int dx over {a emx − be sup −mx} ˜=˜
left { lpile {
     1 over {2 mab} ˜log˜
          {sa emx − sb} over {sa emx + sb}
   above
     1 over mab ˜ tanh sup −1 ( sa over sb emx )
   above
     −1 over mab ˜ coth sup −1 ( sa over sb emx )
}
```

As to ease of construction, we have already mentioned that there are really only a few person-months invested. Much of this time has gone into two things—fine-tuning (what is the most esthetically pleasing space to use between the numerator and denominator of a fraction?), and changing things found deficient by our users (shouldn't a tilde be a delimiter?).

The program consists of a number of small, essentially unconnected modules for code generation, a simple lexical analyzer, a canned parser which we did not have to write, and some miscellany associated with input files and the macro facility. The program is now about 1600 lines of C [6], a high-level language reminiscent of BCPL. About 20 percent of these lines are ''print'' statements, generating the output code.

The semantic routines that generate the actual TROFF commands can be changed to accommodate other formatting languages and devices. For example, in less than 24 hours, one of us changed the entire semantic package to drive NROFF, a variant of TROFF, for typesetting mathematics on teletypewriter devices capable of reverse line motions. Since many potential users do not have access to a typesetter, but still have to type mathematics, this provides a way to get a typed version of the final output which is close enough for debugging purposes, and sometimes even for ultimate use.

7. Conclusions

We think we have shown that it is possible to do acceptably good typesetting of mathematics on a phototypesetter, with an input language that is easy to learn and use and that satisfies many users' demands. Such a package can be implemented in short order, given a compiler-compiler and a decent typesetting program underneath.

Defining a language, and building a compiler for it with a compiler-compiler seems like the only sensible way to do business. Our experience with the use of a grammar and a compiler-compiler has been uniformly favorable. If we had written everything into code directly, we would have been locked into our original design. Furthermore, we would have never been sure where the exceptions and special cases were. But because we have a grammar, we can change our minds readily and still be reasonably sure that if a construction works in one place it will work everywhere.

Acknowledgements

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to modify TROFF to make our task easier and for his continuous assistance during the development of our program. We are also grateful to A. V. Aho for help with language theory, to S. C. Johnson for aid with the compiler-compiler, and to our early users A. V. Aho, S. I. Feldman, S. C. Johnson, R. W. Hamming, and M. D. McIlroy for their constructive criticisms.

References

[1] *A Manual of Style,* 12th Edition. University of Chicago Press, 1969. p 295.

[2] *Model C/A/T Phototypesetter.* Graphic Systems, Inc., Hudson, N. H.

[3] Ritchie, D. M., and Thompson, K. L., ''The UNIX time-sharing system.'' *Comm. ACM 17,* 7 (July 1974), 365-375.

[4] Ossanna, J. F., TROFF User's Manual. Bell Laboratories Computing Science Technical Report 54, 1977.

[5] Aho, A. V., and Johnson, S. C., ''LR Parsing.'' *Comp. Surv. 6,* 2 (June 1974), 99-124.

[6] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Prentice-Hall, Inc., 1978.

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This is the user's guide for a system for typesetting mathematics, using the phototypesetters on the UNIX† and GCOS operating systems.

Mathematical expressions are described in a language designed to be easy to use by people who know neither mathematics nor typesetting. Enough of the language to set in-line expressions like $\lim_{x \to \pi/2} (\tan x)^{\sin 2x} = 1$ or display equations like

$$G(z) = e^{\ln G(z)} = \exp \left(\sum_{k \ge 1} \frac{S_k z^k}{k} \right) = \prod_{k \ge 1} e^{S_k z^k / k}$$

$$= \left(1 + S_1 z + \frac{S_1^2 z^2}{2!} + \cdots \right) \left(1 + \frac{S_2 z^2}{2} + \frac{S_2^2 z^4}{2^2 \cdot 2!} + \cdots \right) \cdots$$

$$= \sum_{m \ge 0} \left(\sum_{\substack{k_1, k_2, \ldots, k_m \ge 0 \\ k_1 + 2k_2 + \cdots + mk_m = m}} \frac{S_1^{k_1}}{1^{k_1} k_1!} \frac{S_2^{k_2}}{2^{k_2} k_2!} \cdots \frac{S_m^{k_m}}{m^{k_m} k_m!} \right) z^m$$

can be learned in an hour or so.

The language interfaces directly with the phototypesetting language TROFF, so mathematical expressions can be embedded in the running text of a manuscript, and the entire document produced in one process. This user's guide is an example of its output.

The same language may be used with the UNIX formatter NROFF to set mathematical expressions on DASI and GSI terminals and Model 37 teletypes.

August 15, 1978

†UNIX is a Trademark of Bell Laboratories.

Typesetting Mathematics — User's Guide (Second Edition)

Brian W. Kernighan and Lorinda L. Cherry

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

EQN is a program for typesetting mathematics on the Graphics Systems phototypesetters on UNIX and GCOS. The EQN language was designed to be easy to use by people who know neither mathematics nor typesetting. Thus EQN knows relatively little about mathematics. In particular, mathematical symbols like +, −, ×, parentheses, and so on have no special meanings. EQN is quite happy to set garbage (but it will look good).

EQN works as a preprocessor for the typesetter formatter, TROFF[1], so the normal mode of operation is to prepare a document with both mathematics and ordinary text interspersed, and let EQN set the mathematics while TROFF does the body of the text.

On UNIX, EQN will also produce mathematics on DASI and GSI terminals and on Model 37 teletypes. The input is identical, but you have to use the programs NEQN and NROFF instead of EQN and TROFF. Of course, some things won't look as good because terminals don't provide the variety of characters, sizes and fonts that a typesetter does, but the output is usually adequate for proofreading.

To use EQN on UNIX,

```
eqn files | troff
```

GCOS use is discussed in section 26.

2. Displayed Equations

To tell EQN where a mathematical expression begins and ends, we mark it with lines beginning .EQ and .EN. Thus if you type the lines

```
.EQ
x=y+z
.EN
```

your output will look like

$$x = y + z$$

The .EQ and .EN are copied through untouched; they are not otherwise processed by EQN. This means that you have to take care of things like centering, numbering, and so on yourself. The most common way is to use the TROFF and NROFF macro package package '−ms' developed by M. E. Lesk[3], which allows you to center, indent, left-justify and number equations.

With the '−ms' package, equations are centered by default. To left-justify an equation, use .EQ L instead of .EQ. To indent it, use .EQ I. Any of these can be followed by an arbitrary 'equation number' which will be placed at the right margin. For example, the input

```
.EQ I (3.1a)
x = f(y/2) + y/2
.EN
```

produces the output

$$x = f\,(y/2) + y/2 \qquad (3.1a)$$

There is also a shorthand notation so in-line expressions like π_i^2 can be entered without .EQ and .EN. We will talk about it in section 19.

3. Input spaces

Spaces and newlines within an expression are thrown away by EQN. (Normal text is left absolutely alone.) Thus between .EQ and .EN,

```
x=y+z
```

and

```
x = y + z
```

and

```
x   =   y
   + z
```

and so on all produce the same output

$$x = y + z$$

You should use spaces and newlines freely to make your input equations readable and easy to edit. In particular, very long lines are a bad idea, since they are often hard to fix if you make a mistake.

4. Output spaces

To force extra spaces into the *output,* use a tilde ''~'' for each space you want:

x~=~y~+~z

gives

$$x\ =\ y\ +\ z$$

You can also use a circumflex ''^'', which gives a space half the width of a tilde. It is mainly useful for fine-tuning. Tabs may also be used to position pieces of an expression, but the tab stops must be set by TROFF commands.

5. Symbols, Special Names, Greek

EQN knows some mathematical symbols, some mathematical names, and the Greek alphabet. For example,

x=2 pi int sin (omega t)dt

produces

$$x=2\pi\int\sin(\omega t)dt$$

Here the spaces in the input are **necessary** to tell EQN that *int, pi, sin* and *omega* are separate entities that should get special treatment. The *sin,* digit 2, and parentheses are set in roman type instead of italic; *pi* and *omega* are made Greek; and *int* becomes the integral sign.

When in doubt, leave spaces around separate parts of the input. A *very* common error is to type *f(pi)* without leaving spaces on both sides of the *pi.* As a result, EQN does not recognize *pi* as a special word, and it appears as $f(pi)$ instead of $f(\pi)$.

A complete list of EQN names appears in section 23. Knowledgeable users can also use TROFF four-character names for anything EQN doesn't know about, like *\(bs* for the Bell System sign.

6. Spaces, Again

The only way EQN can deduce that some sequence of letters might be special is if that sequence is separated from the letters on either side of it. This can be done by surrounding a special word by ordinary spaces (or tabs or newlines), as we did in the previous section.

You can also make special words stand out by surrounding them with tildes or circumflexes:

x~=~2~pi~int~sin~(~omega~t~)~dt

is much the same as the last example, except that the tildes not only separate the magic words like *sin, omega,* and so on, but also add extra spaces, one space per tilde:

$$x\ =\ 2\ \pi\ \int\ \ \sin\ (\ \omega\ t\ \ )\ dt$$

Special words can also be separated by braces { } and double quotes "...", which have special meanings that we will see soon.

7. Subscripts and Superscripts

Subscripts and superscripts are obtained with the words *sub* and *sup.*

x sup 2 + y sub k

gives

$$x^2+y_k$$

EQN takes care of all the size changes and vertical motions needed to make the output look right. The words *sub* and *sup* must be surrounded by spaces; *x sub2* will give you $xsub2$ instead of x_2. Furthermore, don't forget to leave a space (or a tilde, etc.) to mark the end of a subscript or superscript. A common error is to say something like

y = (x sup 2)+1

which causes

$$y=(x^{2)+1}$$

instead of the intended

$$y=(x^2)+1$$

Subscripted subscripts and superscripted superscripts also work:

x sub i sub 1

is

$$x_{i_1}$$

A subscript and superscript on the same thing are printed one above the other if the subscript comes *first:*

x sub i sup 2

is

$$x_i^2$$

Other than this special case, *sub* and *sup* group to the right, so *x sup y sub z* means x^{y_z}, not ${x^y}_z$.

8. Braces for Grouping

Normally, the end of a subscript or superscript is marked simply by a blank (or tab or tilde, etc.) What if the subscript or superscript is something that has to be typed with blanks in it? In that case, you can use the braces { and } to mark the beginning and end of the subscript or superscript:

e sup {i omega t}

is

$$e^{i\omega t}$$

Rule: Braces can *always* be used to force EQN to treat something as a unit, or just to make your intent perfectly clear. Thus:

x sub {i sub 1} sup 2

is

$$x_{i_1}^2$$

with braces, but

x sub i sub 1 sup 2

is

$$x_{i_1^2}$$

which is rather different.

Braces can occur within braces if necessary:

e sup {i pi sup {rho +1}}

is

$$e^{i\pi^{\rho+1}}$$

The general rule is that anywhere you could use some single thing like *x,* you can use an arbitrarily complicated thing if you enclose it in braces. EQN will look after all the details of positioning it and making it the right size.

In all cases, make sure you have the right number of braces. Leaving one out or adding an extra will cause EQN to complain bitterly.

Occasionally you will have to print braces. To do this, enclose them in double quotes, like "{". Quoting is discussed in more detail in section 14.

9. Fractions

To make a fraction, use the word *over:*

a+b over 2c =1

gives

$$\frac{a+b}{2c}=1$$

The line is made the right length and positioned automatically. Braces can be used to make clear what goes over what:

{alpha + beta} over {sin (x)}

is

$$\frac{\alpha+\beta}{\sin(x)}$$

What happens when there is both an *over* and a *sup* in the same expression? In such an apparently ambiguous case, EQN does the *sup* before the *over,* so

−b sup 2 over pi

is $\frac{-b^2}{\pi}$ instead of $-b^{\frac{2}{\pi}}$ The rules which decide which operation is done first in cases like this are summarized in section 23. When in doubt, however, *use braces* to make clear what goes with what.

10. Square Roots

To draw a square root, use *sqrt:*

sqrt a+b + 1 over sqrt {ax sup 2 +bx+c}

is

$$\sqrt{a+b}+\frac{1}{\sqrt{ax^2+bx+c}}$$

Warning — square roots of tall quantities look lousy, because a root-sign big enough to cover the quantity is too dark and heavy:

sqrt {a sup 2 over b sub 2}

is

$$\sqrt{\frac{a^2}{b_2}}$$

Big square roots are generally better written as something to the power ½:

$$(a^2/b_2)^{1/2}$$

which is

(a sup 2 /b sub 2) sup half

11. Summation, Integral, Etc.

Summations, integrals, and similar constructions are easy:

sum from i=0 to {i= inf} x sup i

produces

$$\sum_{i=0}^{i=\infty} x^i$$

Notice that we used braces to indicate where the upper part $i=\infty$ begins and ends. No braces were necessary for the lower part $i=0$, because it contained no blanks. The braces will never hurt, and if the *from* and *to* parts contain any blanks, you must use braces around them.

The *from* and *to* parts are both optional, but if both are used, they have to occur in that order.

Other useful characters can replace the *sum* in our example:

int prod union inter

become, respectively,

$$\int \quad \prod \quad \cup \quad \cap$$

Since the thing before the *from* can be anything, even something in braces, *from-to* can often be used in unexpected ways:

lim from {n –> inf} x sub n =0

is

$$\lim_{n\to\infty} x_n = 0$$

12. Size and Font Changes

By default, equations are set in 10-point type (the same size as this guide), with standard mathematical conventions to determine what characters are in roman and what in italic. Although EQN makes a valiant attempt to use esthetically pleasing sizes and fonts, it is not perfect. To change sizes and fonts, use *size n* and *roman, italic, bold* and *fat.* Like *sub* and *sup,* size and font changes affect only the thing that follows them, and revert to the normal situation at the end of it. Thus

bold x y

is

$$\mathbf{x}y$$

and

size 14 bold x = y +
size 14 {alpha + beta}

gives

$$\mathbf{x}=y+\alpha+\beta$$

As always, you can use braces if you want to affect something more complicated than a single letter. For example, you can change the size of an entire equation by

size 12 { ... }

Legal sizes which may follow *size* are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, 36. You can also change the size *by* a given amount; for example, you can say *size +2* to make the size two points bigger, or *size −3* to make it three points smaller. This has the advantage that you don't have to know what the current size is.

If you are using fonts other than roman, italic and bold, you can say *font X* where *X* is a one character TROFF name or number for the font. Since EQN is tuned for roman, italic and bold, other fonts may not give quite as good an appearance.

The *fat* operation takes the current font and widens it by overstriking: *fat grad* is $\boldsymbol{\nabla}$ and *fat {x sub i}* is $\boldsymbol{x_i}$.

If an entire document is to be in a non-standard size or font, it is a severe nuisance to have to write out a size and font change for each equation. Accordingly, you can set a ''global'' size or font which thereafter affects all equations. At the beginning of any equation, you might say, for instance,

```
.EQ
gsize 16
gfont R
 ...
.EN
```

to set the size to 16 and the font to roman thereafter. In place of R, you can use any of the TROFF font names. The size after *gsize* can be a relative change with + or −.

Generally, *gsize* and *gfont* will appear at the beginning of a document but they can also appear thoughout a document: the global font and size can be changed as often as needed. For example, in a footnote‡ you will typically want the size of equations to match the size of the footnote text, which is two points smaller than the main text. Don't forget to reset the global size at the end of the footnote.

13. Diacritical Marks

To get funny marks on top of letters, there are several words:

x dot	$\dot{x}$
x dotdot	$\ddot{x}$
x hat	$\hat{x}$
x tilde	$\tilde{x}$
x vec	$\vec{x}$
x dyad	$\overleftrightarrow{x}$
x bar	$\bar{x}$
x under	$\underline{x}$

The diacritical mark is placed at the right height. The *bar* and *under* are made the right length for the entire construct, as in $\overline{x+y+z}$; other marks are centered.

14. Quoted Text

Any input entirely within quotes ("...") is not subject to any of the font changes and spacing adjustments normally done by the equation setter. This provides a way to do your own spacing and adjusting if needed:

italic "sin(x)" + sin (x)

is

$$sin(x)+\sin(x)$$

Quotes are also used to get braces and other EQN keywords printed:

"{ size alpha }"

is

{ *size alpha* }

and

roman "{ size alpha }"

is

{ size alpha }

The construction "" is often used as a place-holder when grammatically EQN needs something, but you don't actually want anything in your output. For example, to make ${}^{2}\mathrm{He}$, you can't just type *sup 2 roman He* because a *sup* has to be a superscript *on* something. Thus you must say

"" sup 2 roman He

To get a literal quote use ``\"''. TROFF characters like \(*bs* can appear unquoted, but more complicated things like horizontal and vertical motions with *h* and *v* should always be quoted. (If you've never heard of *h* and *v*, ignore this section.)

15. Lining Up Equations

Sometimes it's necessary to line up a series of equations at some horizontal position, often at an equals sign. This is done with two operations called *mark* and *lineup*.

The word *mark* may appear once at any place in an equation. It remembers the horizontal position where it appeared. Successive equations can contain one occurrence of the word *lineup*. The place where *lineup* appears is made to line up with the place marked by the previous *mark* if at all possible. Thus, for example, you can say

```
.EQ I
x+y mark = z
.EN
.EQ I
x lineup = 1
.EN
```

to produce

$$x+y=z$$

$$x=1$$

For reasons too complicated to talk about, when you use EQN and '–ms', use either .EQ I or .EQ L. mark and *lineup* don't work with centered equations. Also bear in mind that *mark* doesn't look ahead;

```
x mark =1
 ...
x+y lineup =z
```

isn't going to work, because there isn't room for the $x+y$ part after the *mark* remembers where the x is.

‡Like this one, in which we have a few random expressions like x_i and π^2. The sizes for these were set by the command *gsize –2*.

16. Big Brackets, Etc.

To get big brackets [ ], braces { }, parentheses ( ), and bars | | around things, use the *left* and *right* commands:

```
left { a over b + 1 right }
 ~=~ left ( c over d right )
 + left [ e right ]
```

is

$$\left\{ \frac{a}{b}+1 \right\} = \left(\frac{c}{d} \right) + \left[e \right]$$

The resulting brackets are made big enough to cover whatever they enclose. Other characters can be used besides these, but the are not likely to look very good. One exception is the *floor* and *ceiling* characters:

```
left floor x over y right floor
<= left ceiling a over b right ceiling
```

produces

$$\left\lfloor \frac{x}{y} \right\rfloor \le \left\lceil \frac{a}{b} \right\rceil$$

Several warnings about brackets are in order. First, braces are typically bigger than brackets and parentheses, because they are made up of three, five, seven, etc., pieces, while brackets can be made up of two, three, etc. Second, big left and right parentheses often look poor, because the character set is poorly designed.

The *right* part may be omitted: a ''left something'' need not have a corresponding ''right something''. If the *right* part is omitted, put braces around the thing you want the left bracket to encompass. Otherwise, the resulting brackets may be too large.

If you want to omit the *left* part, things are more complicated, because technically you can't have a *right* without a corresponding *left.* Instead you have to say

```
left "" ..... right )
```

for example. The *left* "" means a ''left nothing''. This satisfies the rules without hurting your output.

17. Piles

There is a general facility for making vertical piles of things; it comes in several flavors. For example:

```
A ~=~ left [
  pile { a above b above c }
  ~~ pile { x above y above z }
right ]
```

will make

$$A = \left[\begin{matrix} a & x \\ b & y \\ c & z \end{matrix} \right]$$

The elements of the pile (there can be as many as you want) are centered one above another, at the right height for most purposes. The keyword *above* is used to separate the pieces; braces are used around the entire list. The elements of a pile can be as complicated as needed, even containing more piles.

Three other forms of pile exist: *lpile* makes a pile with the elements left-justified; *rpile* makes a right-justified pile; and *cpile* makes a centered pile, just like *pile.* The vertical spacing between the pieces is somewhat larger for *l-, r-* and *cpiles* than it is for ordinary piles.

```
roman sign (x)~=~
left {
   lpile {1 above 0 above -1}
   ~~ lpile
    {if~x>0 above if~x=0 above if~x<0}
```

makes

$$\mathrm{sign}(x) = \left\{ \begin{array}{ll} 1 & \text{if } x>0 \\ 0 & \text{if } x=0 \\ -1 & \text{if } x<0 \end{array} \right.$$

Notice the left brace without a matching right one.

18. Matrices

It is also possible to make matrices. For example, to make a neat array like

$$\begin{matrix} x_i & x^2 \\ y_i & y^2 \end{matrix}$$

you have to type

```
matrix {
  ccol { x sub i above y sub i }
  ccol { x sup 2 above y sup 2 }
}
```

This produces a matrix with two centered columns. The elements of the columns are then listed just as for a pile, each element separated by the word *above.* You can also use *lcol* or *rcol* to left or right adjust columns. Each column can be separately adjusted, and there can be as many columns as you like.

The reason for using a matrix instead of two adjacent piles, by the way, is that if the elements of the piles don't all have the same height, they won't line up properly. A matrix forces them to line up, because it looks at the entire structure before deciding what spacing to use.

A word of warning about matrices — *each column must have the same number of elements in it.* The world will end if you get this wrong.

19. Shorthand for In-line Equations

In a mathematical document, it is necessary to follow mathematical conventions not just in display equations, but also in the body of the text, for example by making variable names like x italic. Although this could be done by surrounding the appropriate parts with .EQ and .EN, the continual repetition of .EQ and .EN is a nuisance. Furthermore, with '–ms', .EQ and .EN imply a displayed equation.

EQN provides a shorthand for short in-line expressions. You can define two characters to mark the left and right ends of an in-line equation, and then type expressions right in the middle of text lines. To set both the left and right characters to dollar signs, for example, add to the beginning of your document the three lines

```
.EQ
delim $$
.EN
```

Having done this, you can then say things like

> Let $alpha sub i$ be the primary variable, and let $beta$ be zero. Then we can show that $x sub 1$ is $>=0$.

This works as you might expect — spaces, newlines, and so on are significant in the text, but not in the equation part itself. Multiple equations can occur in a single input line.

Enough room is left before and after a line that contains in-line expressions that something like $\sum_{i=1}^{n} x_i$ does not interfere with the lines surrounding it.

To turn off the delimiters,

```
.EQ
delim off
.EN
```

Warning: don't use braces, tildes, circumflexes, or double quotes as delimiters — chaos will result.

20. Definitions

EQN provides a facility so you can give a frequently-used string of characters a name, and thereafter just type the name instead of the whole string. For example, if the sequence

```
x sub i sub 1 + y sub i sub 1
```

appears repeatedly throughout a paper, you can save re-typing it each time by defining it like this:

```
define xy ′x sub i sub 1 + y sub i sub 1′
```

This makes *xy* a shorthand for whatever characters occur between the single quotes in the definition. You can use any character instead of quote to mark the ends of the definition, so long as it doesn't appear inside the definition.

Now you can use *xy* like this:

```
.EQ
f(x) = xy ...
.EN
```

and so on. Each occurrence of *xy* will expand into what it was defined as. Be careful to leave spaces or their equivalent around the name when you actually use it, so EQN will be able to identify it as special.

There are several things to watch out for. First, although definitions can use previous definitions, as in

```
.EQ
define xi ′ x sub i ′
define xi1 ′ xi sub 1 ′
.EN
```

don't define something in terms of itself' A favorite error is to say

```
define X ′ roman X ′
```

This is a guaranteed disaster, since X *is* now defined in terms of itself. If you say

```
define X ′ roman "X" ′
```

however, the quotes protect the second X, and everything works fine.

EQN keywords can be redefined. You can make / mean *over* by saying

define / ′ over ′

or redefine *over* as / with

define over ′ / ′

If you need different things to print on a terminal and on the typesetter, it is sometimes worth defining a symbol differently in NEQN and EQN. This can be done with *ndefine* and *tdefine.* A definition made with *ndefine* only takes effect if you are running NEQN; if you use *tdefine,* the definition only applies for EQN. Names defined with plain *define* apply to both EQN and NEQN.

21. Local Motions

Although EQN tries to get most things at the right place on the paper, it isn't perfect, and occasionally you will need to tune the output to make it just right. Small extra horizontal spaces can be obtained with tilde and circumflex. You can also say *back n* and *fwd n* to move small amounts horizontally. *n* is how far to move in 1/100's of an em (an em is about the width of the letter 'm'.) Thus *back 50* moves back about half the width of an m. Similarly you can move things up or down with *up n* and *down n.* As with *sub* or *sup,* the local motions affect the next thing in the input, and this can be something arbitrarily complicated if it is enclosed in braces.

22. A Large Example

Here is the complete source for the three display equations in the abstract of this guide.

```
.EQ I
G(z)~mark =~ e sup { ln ~ G(z) }
~=~ exp left (
sum from k>=1 {S sub k z sup k} over k right )
~=~  prod from k>=1 e sup {S sub k z sup k /k}
.EN
.EQ I
lineup = left ( 1 + S sub 1 z +
{ S sub 1 sup 2 z sup 2 } over 2! + ... right )
left ( 1+ { S sub 2 z sup 2 } over 2
+ { S sub 2 sup 2 z sup 4 } over { 2 sup 2 cdot 2! }
+ ... right ) ...
.EN
.EQ I
lineup =  sum from m>=0 left (
sum from
pile { k sub 1 ,k sub 2 ,..., k sub m  >=0
above
k sub 1 +2k sub 2 + ... +mk sub m =m}
{ S sub 1 sup {k sub 1} } over {1 sup k sub 1 k sub 1 ! } ~
{ S sub 2 sup {k sub 2} } over {2 sup k sub 2 k sub 2 ! } ~
...
{ S sub m sup {k sub m} } over {m sup k sub m k sub m ! }
right ) z sup m
.EN
```

23. Keywords, Precedences, Etc.

If you don't use braces, EQN will do operations in the order shown in this list.

dyad vec under bar tilde hat dot dotdot
fwd back down up
fat roman italic bold size
sub sup sqrt over
from to

These operations group to the left:

over sqrt left right

All others group to the right.

Digits, parentheses, brackets, punctuation marks, and these mathematical words are converted to Roman font when encountered:

sin cos tan sinh cosh tanh arc
max min lim log ln exp
Re Im and if for det

These character sequences are recognized and translated as shown.

>=	≥
<=	≤
==	≡
!=	≠
+−	±
−>	→
<−	←
<<	≪
>>	≫
inf	∞
partial	∂
half	½
prime	′
approx	≈
nothing	
cdot	·
times	×
del	∇
grad	∇
...	· · ·
,...,	, . . . ,
sum	Σ
int	∫
prod	Π

union	$\cup$
inter	$\cap$

To obtain Greek letters, simply spell them out in whatever case you want:

DELTA	Δ	iota	ι
GAMMA	Γ	kappa	κ
LAMBDA	Λ	lambda	λ
OMEGA	Ω	mu	μ
PHI	Φ	nu	ν
PI	Π	omega	ω
PSI	Ψ	omicron	o
SIGMA	Σ	phi	ϕ
THETA	Θ	pi	π
UPSILON	Υ	psi	ψ
XI	Ξ	rho	ρ
alpha	α	sigma	σ
beta	β	tau	τ
chi	χ	theta	θ
delta	δ	upsilon	υ
epsilon	ϵ	xi	ξ
eta	η	zeta	ζ
gamma	γ		

These are all the words known to EQN (except for characters with names), together with the section where they are discussed.

above	17, 18	lpile	17
back	21	mark	15
bar	13	matrix	18
bold	12	ndefine	20
ccol	18	over	9
col	18	pile	17
cpile	17	rcol	18
define	20	right	16
delim	19	roman	12
dot	13	rpile	17
dotdot	13	size	12
down	21	sqrt	10
dyad	13	sub	7
fat	12	sup	7
font	12	tdefine	20
from	11	tilde	13
fwd	21	to	11
gfont	12	under	13
gsize	12	up	21
hat	13	vec	13
italic	12	~, ^	4, 6
lcol	18	{ }	8
left	16	"..."	8, 14
lineup	15		

24. Troubleshooting

If you make a mistake in an equation, like leaving out a brace (very common) or having one too many (very common) or having a *sup* with nothing before it (common), EQN will tell you with the message

syntax error between lines x and y, file z

where x and y are approximately the lines between which the trouble occurred, and z is the name of the file in question. The line numbers are approximate — look nearby as well. There are also self-explanatory messages that arise if you leave out a quote or try to run EQN on a non-existent file.

If you want to check a document before actually printing it (on UNIX only),

```
eqn  files >/dev/null
```

will throw away the output but print the messages.

If you use something like dollar signs as delimiters, it is easy to leave one out. This causes very strange troubles. The program *checkeq* (on GCOS, use *./checkeq* instead) checks for misplaced or missing dollar signs and similar troubles.

In-line equations can only be so big because of an internal buffer in TROFF. If you get a message ``word overflow'', you have exceeded this limit. If you print the equation as a displayed equation this message will usually go away. The message ``line overflow'' indicates you have exceeded an even bigger buffer. The only cure for this is to break the equation into two separate ones.

On a related topic, EQN does not break equations by itself — you must split long equations up across multiple lines by yourself, marking each by a separate .EQ ... .EN sequence. EQN does warn about equations that are too long to fit on one line.

25. Use on UNIX

To print a document that contains mathematics on the UNIX typesetter,

```
eqn files | troff
```

If there are any TROFF options, they go after the TROFF part of the command. For example,

```
eqn files | troff -ms
```

To run the same document on the GCOS

typesetter, use

eqn files | troff −g (other options) | gcat

A compatible version of EQN can be used on devices like teletypes and DASI and GSI terminals which have half-line forward and reverse capabilities. To print equations on a Model 37 teletype, for example, use

neqn files | nroff

The language for equations recognized by NEQN is identical to that of EQN, although of course the output is more restricted.

To use a GSI or DASI terminal as the output device,

neqn files | nroff −T*x*

where *x* is the terminal type you are using, such as *300* or *300S.*

EQN and NEQN can be used with the TBL program[2] for setting tables that contain mathematics. Use TBL before [N]EQN, like this:

tbl files | eqn | troff
tbl files | neqn | nroff

26. Acknowledgments

We are deeply indebted to J. F. Ossanna, the author of TROFF, for his willingness to extend TROFF to make our task easier, and for his continuous assistance during the development and evolution of EQN. We are also grateful to A. V. Aho for advice on language design, to S. C. Johnson for assistance with the YACC compiler-compiler, and to all the EQN users who have made helpful suggestions and criticisms.

References

[1] J. F. Ossanna, ‘‘NROFF/TROFF User's Manual’’, Bell Laboratories Computing Science Technical Report #54, 1976.

[2] M. E. Lesk, ‘‘Typing Documents on UNIX’’, Bell Laboratories, 1976.

[3] M. E. Lesk, ‘‘TBL — A Program for Setting Tables’’, Bell Laboratories Computing Science Technical Report #49, 1976.

Tbl — A Program to Format Tables

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Tbl is a document formatting preprocessor for *troff* or *nroff* which makes even fairly complex tables easy to specify and enter. It is available on the PDP-11 UNIX* system and on Honeywell 6000 GCOS. Tables are made up of columns which may be independently centered, right-adjusted, left-adjusted, or aligned by decimal points. Headings may be placed over single columns or groups of columns. A table entry may contain equations, or may consist of several rows of text. Horizontal or vertical lines may be drawn as desired in the table, and any table or element may be enclosed in a box. For example:

1970 Federal Budget Transfers (in billions of dollars)			
State	Taxes collected	Money spent	Net
New York	22.91	21.35	−1.56
New Jersey	8.33	6.96	−1.37
Connecticut	4.12	3.10	−1.02
Maine	0.74	0.67	−0.07
California	22.29	22.42	+0.13
New Mexico	0.70	1.49	+0.79
Georgia	3.30	4.28	+0.98
Mississippi	1.15	2.32	+1.17
Texas	9.33	11.13	+1.80

January 16, 1979

* UNIX is a Trademark/Service Mark of the Bell System

Tbl — A Program to Format Tables

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction.

Tbl turns a simple description of a table into a *troff* or *nroff* [1] program (list of commands) that prints the table. *Tbl* may be used on the PDP-11 UNIX [2] system and on the Honeywell 6000 GCOS system. It attempts to isolate a portion of a job that it can successfully handle and leave the remainder for other programs. Thus *tbl* may be used with the equation formatting program *eqn* [3] or various layout macro packages [4,5,6], but does not duplicate their functions.

This memorandum is divided into two parts. First we give the rules for preparing *tbl* input; then some examples are shown. The description of rules is precise but technical, and the beginning user may prefer to read the examples first, as they show some common table arrangements. A section explaining how to invoke *tbl* precedes the examples. To avoid repetition, henceforth read *troff* as *''troff* or *nroff.''*

The input to *tbl* is text for a document, with tables preceded by a ''.TS'' (table start) command and followed by a ''.TE'' (table end) command. *Tbl* processes the tables, generating *troff* formatting commands, and leaves the remainder of the text unchanged. The ''.TS'' and ''.TE'' lines are copied, too, so that *troff* page layout macros (such as the memo formatting macros [4]) can use these lines to delimit and place tables as they see fit. In particular, any arguments on the ''.TS'' or ''.TE'' lines are copied but otherwise ignored, and may be used by document layout macro commands.

The format of the input is as follows:

text
.TS
table
.TE
text
.TS
table
.TE
text
. . .

where the format of each table is as follows:

.TS
options **;**
format **.**
data
.TE

Each table is independent, and must contain formatting information followed by the data to be entered in the table. The formatting information, which describes the individual columns and rows of the table, may be preceded by a few options that affect the entire table. A detailed description of tables is given in the next section.

Input commands.

As indicated above, a table contains, first, global options, then a format section describing the layout of the table entries, and then the data to be printed. The format and data are always required, but not the options. The various parts of the table are entered as follows:

1) OPTIONS. There may be a single line of options affecting the whole table. If present, this line must follow the **.**TS line immediately and must contain a list of option names separated by spaces, tabs, or commas, and must be terminated by a semicolon. The allowable options are:

center	— center the table (default is left-adjust);
expand	— make the table as wide as the current line length;
box	— enclose the table in a box;
allbox	— enclose each item in the table in a box;
doublebox	— enclose the table in two boxes;
tab (*x*)	— use *x* instead of tab to separate data items.
linesize (*n*)	— set lines or rules (e.g. from **box**) in *n* point type;
delim (*xy*)	— recognize *x* and *y* as the *eqn* delimiters.

The *tbl* program tries to keep boxed tables on one page by issuing appropriate ''need'' (***.ne***) commands. These requests are calculated from the number of lines in the tables, and if there are spacing commands embedded in the input, these requests may be inaccurate; use normal *troff* procedures, such as keep-release macros, in that case. The user who must have a multi-page boxed table should use macros designed for this purpose, as explained below under 'Usage.'

2) FORMAT. The format section of the table specifies the layout of the columns. Each line in this section corresponds to one line of the table (except that the last line corresponds to all following lines up to the next **.**T&, if any — see below), and each line contains a key-letter for each column of the table. It is good practice to separate the key letters for each column by spaces or tabs. Each key-letter is one of the following:

L or **l**	to indicate a left-adjusted column entry;
R or **r**	to indicate a right-adjusted column entry;
C or **c**	to indicate a centered column entry;
N or **n**	to indicate a numerical column entry, to be aligned with other numerical entries so that the units digits of numbers line up;
A or **a**	to indicate an alphabetic subcolumn; all corresponding entries are aligned on the left, and positioned so that the widest is centered within the column (see example on page 12);
S or **s**	to indicate a spanned heading, i.e. to indicate that the entry from the previous column continues across this column (not allowed for the first column, obviously); or
^	to indicate a vertically spanned heading, i.e. to indicate that the entry from the previous row continues down through this row. (Not allowed for the first row of the table, obviously).

When numerical alignment is specified, a location for the decimal point is sought. The rightmost dot (**.**) adjacent to a digit is used as a decimal point; if there is no dot adjoining a digit, the rightmost digit is used as a units digit; if no alignment is indicated, the item is centered in the column. However, the special non-printing character string \& may be used to override unconditionally dots and digits, or to align alphabetic data; this string lines up where a dot normally would, and then disappears from the final output. In the example below, the items shown at the left will be aligned (in a numerical column) as shown on the right:

13	13
4.2	4.2
26.4.12	26.4.12
abc	abc
abc\&	abc
43\&3.22	433.22
749.12	749.12

Note: If numerical data are used in the same column with wider **L** or **r** type table entries, the widest *number* is centered relative to the wider **L** or **r** items (**L** is used instead of **l** for readability; they have the same meaning as key-letters). Alignment within the numerical items is preserved. This is similar to the behavior of **a** type data, as explained above. However, alphabetic subcolumns (requested by the **a** key-letter) are always slightly indented relative to **L** items; if necessary, the column width is increased to force this. This is not true for **n** type entries.

Warning: the **n** and **a** items should not be used in the same column.

For readability, the key-letters describing each column should be separated by spaces. The end of the format section is indicated by a period. The layout of the key-letters in the format section resembles the layout of the actual data in the table. Thus a simple format might appear as:

```
c s s
l n n .
```

which specifies a table of three columns. The first line of the table contains a heading centered across all three columns; each remaining line contains a left-adjusted item in the first column followed by two columns of numerical data. A sample table in this format might be:

Overall title		
Item-a	34.22	9.1
Item-b	12.65	.02
Items: c,d,e	23	5.8
Total	69.87	14.92

There are some additional features of the key-letter system:

Horizontal lines — A key-letter may be replaced by '_' (underscore) to indicate a horizontal line in place of the corresponding column entry, or by '=' to indicate a double horizontal line. If an adjacent column contains a horizontal line, or if there are vertical lines adjoining this column, this horizontal line is extended to meet the nearby lines. If any data entry is provided for this column, it is ignored and a warning message is printed.

Vertical lines — A vertical bar may be placed between column key-letters. This will cause a vertical line between the corresponding columns of the table. A vertical bar to the left of the first key-letter or to the right of the last one produces a line at the edge of the table. If two vertical bars appear between key-letters, a double vertical line is drawn.

Space between columns — A number may follow the key-letter. This indicates the amount of separation between this column and the next column. The number normally specifies the separation in *ens* (one en is about the width of the letter 'n').* If the ''expand'' option is used, then these numbers are multiplied by a constant such that the table is as wide as the current line length. The default column separation number is 3. If the separation is changed the worst case (largest space requested) governs.

Vertical spanning — Normally, vertically spanned items extending over several rows of the table are centered in their vertical range. If a key-letter is followed by **t** or **T**, any corresponding vertically spanned item will begin at the top line of its range.

* More precisely, an en is a number of points (1 point = 1/72 inch) equal to half the current type size.

Font changes — A key-letter may be followed by a string containing a font name or number preceded by the letter **f** or **F**. This indicates that the corresponding column should be in a different font from the default font (usually Roman). All font names are one or two letters; a one-letter font name should be separated from whatever follows by a space or tab. The single letters **B**, **b**, **I**, and **i** are shorter synonyms for **fB** and **fI**. Font change commands given with the table entries override these specifications.

Point size changes — A key-letter may be followed by the letter **p** or **P** and a number to indicate the point size of the corresponding table entries. The number may be a signed digit, in which case it is taken as an increment or decrement from the current point size. If both a point size and a column separation value are given, one or more blanks must separate them.

Vertical spacing changes — A key-letter may be followed by the letter **v** or **V** and a number to indicate the vertical line spacing to be used within a multi-line corresponding table entry. The number may be a signed digit, in which case it is taken as an increment or decrement from the current vertical spacing. A column separation value must be separated by blanks or some other specification from a vertical spacing request. This request has no effect unless the corresponding table entry is a text block (see below).

Column width indication — A key-letter may be followed by the letter **w** or **W** and a width value in parentheses. This width is used as a minimum column width. If the largest element in the column is not as wide as the width value given after the **w**, the largest element is assumed to be that wide. If the largest element in the column is wider than the specified value, its width is used. The width is also used as a default line length for included text blocks. Normal *troff* units can be used to scale the width value; if none are used, the default is ens. If the width specification is a unitless integer the parentheses may be omitted. If the width value is changed in a column, the *last* one given controls.

Equal width columns — A key-letter may be followed by the letter **e** or **E** to indicate equal width columns. All columns whose key-letters are followed by **e** or **E** are made the same width. This permits the user to get a group of regularly spaced columns.

Note: The order of the above features is immaterial; they need not be separated by spaces, except as indicated above to avoid ambiguities involving point size and font changes. Thus a numerical column entry in italic font and 12 point type with a minimum width of 2.5 inches and separated by 6 ens from the next column could be specified as

```
np12w(2.5i)fI 6
```

Alternative notation — Instead of listing the format of successive lines of a table on consecutive lines of the format section, successive line formats may be given on the same line, separated by commas, so that the format for the example above might have been written:

```
c s s, l n n .
```

Default — Column descriptors missing from the end of a format line are assumed to be **L**. The longest line in the format section, however, defines the number of columns in the table; extra columns in the data are ignored silently.

3) DATA. The data for the table are typed after the format. Normally, each table line is typed as one line of data. Very long input lines can be broken: any line whose last character is \ is combined with the following line (and the \ vanishes). The data for different columns (the table entries) are separated by tabs, or by whatever character has been specified in the option *tabs* option. There are a few special cases:

Troff commands within tables — An input line beginning with a '.' followed by anything but a number is assumed to be a command to *troff* and is passed through unchanged, retaining its position in the table. So, for example, space within a table may be produced by ''.sp'' commands in the data.

Full width horizontal lines — An input *line* containing only the character _ (underscore) or **=** (equal sign) is taken to be a single or double line, respectively, extending the full width of the *table.*

Single column horizontal lines — An input table *entry* containing only the character _ or **=** is taken to be a single or double line extending the full width of the *column.* Such lines are extended to meet horizontal or vertical lines adjoining this column. To obtain these characters explicitly in a column, either precede them by \& or follow them by a space before the usual tab or newline.

Short horizontal lines — An input table *entry* containing only the string **_** is taken to be a single line as wide as the contents of the column. It is not extended to meet adjoining lines.

Repeated characters — An input table *entry* containing only a string of the form **\R***x* where *x* is any character is replaced by repetitions of the character *x* as wide as the data in the column. The sequence of *x*'s is not extended to meet adjoining columns.

Vertically spanned items — An input table entry containing only the character string \^ indicates that the table entry immediately above spans downward over this row. It is equivalent to a table format key-letter of '^'.

Text blocks — In order to include a block of text as a table entry, precede it by **T{** and follow it by **T}**. Thus the sequence

. . . T{
block of
text
T} . . .

is the way to enter, as a single entry in the table, something that cannot conveniently be typed as a simple string between tabs. Note that the **T}** end delimiter must begin a line; additional columns of data may follow after a tab on the same line. See the example on page 10 for an illustration of included text blocks in a table. If more than twenty or thirty text blocks are used in a table, various limits in the *troff* program are likely to be exceeded, producing diagnostics such as 'too many string/macro names' or 'too many number registers.'

Text blocks are pulled out from the table, processed separately by *troff,* and replaced in the table as a solid block. If no line length is specified in the *block of text* itself, or in the table format, the default is to use $L \times C/(N+1)$ where L is the current line length, C is the number of table columns spanned by the text, and N is the total number of columns in the table. The other parameters (point size, font, etc.) used in setting the *block of text* are those in effect at the beginning of the table (including the effect of the ''.TS'' macro) and any table format specifications of size, spacing and font, using the **p**, **v** and **f** modifiers to the column key-letters. Commands within the text block itself are also recognized, of course. However, *troff* commands within the table data but not within the text block do not affect that block.

Warnings: — Although any number of lines may be present in a table, only the first 200 lines are used in calculating the widths of the various columns. A multi-page table, of course, may be arranged as several single-page tables if this proves to be a problem. Other difficulties with formatting may arise because, in the calculation of column widths all table entries are assumed to be in the font and size being used when the ''.TS'' command was encountered, except for font and size changes indicated (a) in the table format section and (b) within the table data (as in the entry \s+3\fIdata\fP\s0). Therefore, although arbitrary *troff* requests may be sprinkled in a table, care must be taken to avoid confusing the width calculations; use requests such as '.ps' with care.

4) ADDITIONAL COMMAND LINES. If the format of a table must be changed after many similar lines, as with sub-headings or summarizations, the ''.T&'' (table continue) command can be used to change column parameters. The outline of such a table input is:

```
.TS
options ;
format .
data
. . .
.T&
format .
data
.T&
format .
data
.TE
```

as in the examples on pages 10 and 12. Using this procedure, each table line can be close to its corresponding format line.

Warning: it is not possible to change the number of columns, the space between columns, the global options such as *box,* or the selection of columns to be made equal width.

Usage.

On UNIX, *tbl* can be run on a simple table with the command

tbl input-file | troff

but for more complicated use, where there are several input files, and they contain equations and *ms* memorandum layout commands as well as tables, the normal command would be

tbl file-1 file-2 . . . | eqn | troff −ms

and, of course, the usual options may be used on the *troff* and *eqn* commands. The usage for *nroff* is similar to that for *troff,* but only TELETYPE® Model 37 and Diablo-mechanism (DASI or GSI) terminals can print boxed tables directly.

For the convenience of users employing line printers without adequate driving tables or post-filters, there is a special −*TX* command line option to *tbl* which produces output that does not have fractional line motions in it. The only other command line options recognized by *tbl* are −*ms* and −*mm* which are turned into commands to fetch the corresponding macro files; usually it is more convenient to place these arguments on the *troff* part of the command line, but they are accepted by *tbl* as well.

Note that when *eqn* and *tbl* are used together on the same file *tbl* should be used first. If there are no equations within tables, either order works, but it is usually faster to run *tbl* first, since *eqn* normally produces a larger expansion of the input than *tbl.* However, if there are equations within tables (using the *delim* mechanism in *eqn*), *tbl* must be first or the output will be scrambled. Users must also beware of using equations in **n**-style columns; this is nearly always wrong, since *tbl* attempts to split numerical format items into two parts and this is not possible with equations. The user can defend against this by giving the *delim(xx)* table option; this prevents splitting of numerical columns within the delimiters. For example, if the *eqn* delimiters are *$$*, giving *delim($$)* a numerical column such as ‘‘1245 $+- 16$’’ will be divided after 1245, not after 16.

Tbl limits tables to twenty columns; however, use of more than 16 numerical columns may fail because of limits in *troff,* producing the ‘too many number registers’ message. *Troff* number registers used by *tbl* must be avoided by the user within tables; these include two-digit names from 31 to 99, and names of the forms #*x*, *x*+, *x* |, ^*x*, and *x*–, where *x* is any lower case letter. The names ##, #–, and #^ are also used in certain circumstances. To conserve number register names, the **n** and **a** formats share a register; hence the restriction above that they may not be used in the same column.

For aid in writing layout macros, *tbl* defines a number register TW which is the table width; it is defined by the time that the ‘‘.TE’’ macro is invoked and may be used in the expansion of that macro. More importantly, to assist in laying out multi-page boxed tables the macro T# is defined to produce the bottom lines and side lines of a boxed table, and then invoked at its end. By use of this macro in the page footer a multi-page table can be boxed. In particular, the *ms* macros can be used to print a multi-page boxed table with a repeated heading by giving the argument H to the ‘‘.TS’’ macro. If the table start macro is written

```
.TS H
```

a line of the form

```
.TH
```

must be given in the table after any table heading (or at the start if none). Material up to the ‘‘.TH’’ is placed at the top of each page of table; the remaining lines in the table are placed on several pages as required. Note that this is *not* a feature of *tbl,* but of the *ms* layout macros.

Examples.

Here are some examples illustrating features of *tbl.* The symbol ⓣ in the input represents a tab character.

Input:

```
.TS
box;
c c c
l l l.
LanguageⓣAuthorsⓣRuns on

FortranⓣManyⓣAlmost anything
PL/1ⓣIBMⓣ360/370
CⓣBTLⓣ11/45,H6000,370
BLISSⓣCarnegie-MellonⓣPDP-10,11
IDSⓣHoneywellⓣH6000
PascalⓣStanfordⓣ370
.TE
```

Output:

Language	Authors	Runs on
Fortran	Many	Almost anything
PL/1	IBM	360/370
C	BTL	11/45,H6000,370
BLISS	Carnegie-Mellon	PDP-10,11
IDS	Honeywell	H6000
Pascal	Stanford	370

Input:

```
.TS
allbox;
c s s
c c c
n n n.
AT&T Common Stock
YearⓣPriceⓣDividend
1971ⓣ41-54ⓣ$2.60
2ⓣ41-54ⓣ2.70
3ⓣ46-55ⓣ2.87
4ⓣ40-53ⓣ3.24
5ⓣ45-52ⓣ3.40
6ⓣ51-59ⓣ.95*
.TE
* (first quarter only)
```

Output:

AT&T Common Stock		
Year	Price	Dividend
1971	41-54	$2.60
2	41-54	2.70
3	46-55	2.87
4	40-53	3.24
5	45-52	3.40
6	51-59	.95*

* (first quarter only)

Input:

```
.TS
box;
c s s
c | c | c
l | l | n.
Major New York Bridges
=
BridgeⓉDesignerⓉLength
_
BrooklynⓉJ. A. RoeblingⓉ1595
ManhattanⓉG. LindenthalⓉ1470
WilliamsburgⓉL. L. BuckⓉ1600
_
QueensboroughⓉPalmer &Ⓣ1182
Ⓣ   Hornbostel
_
ⓉⓉ1380
TriboroughⓉO. H. AmmannⓉ_
ⓉⓉ383
_
Bronx WhitestoneⓉO. H. AmmannⓉ2300
Throgs NeckⓉO. H. AmmannⓉ1800
_
George WashingtonⓉO. H. AmmannⓉ3500
.TE
```

Output:

Major New York Bridges		
Bridge	Designer	Length
Brooklyn	J. A. Roebling	1595
Manhattan	G. Lindenthal	1470
Williamsburg	L. L. Buck	1600
Queensborough	Palmer & Hornbostel	1182
		1380
Triborough	O. H. Ammann	
		383
Bronx Whitestone	O. H. Ammann	2300
Throgs Neck	O. H. Ammann	1800
George Washington	O. H. Ammann	3500

Input:

```
.TS
c c
np-2 | n | .
ⓉStack
Ⓣ_
1Ⓣ46
Ⓣ_
2Ⓣ23
Ⓣ_
3Ⓣ15
Ⓣ_
4Ⓣ6.5
Ⓣ_
5Ⓣ2.1
Ⓣ_
.TE
```

Output:

	Stack
1	46
2	23
3	15
4	6.5
5	2.1

Input:

```
.TS
box;
L L L
L L _
L L | LB
L L _
L L L.
januaryⓣfebruaryⓣmarch
aprilⓣmay
juneⓣjulyⓣMonths
augustⓣseptember
octoberⓣnovemberⓣdecember
.TE
```

Output:

january	february	march
april	may	
june	july	**Months**
august	september	
october	november	december

Input:

```
.TS
box;
cfB s s s.
Composition of Foods
_
.T&
c  | c s s
c  | c s s
c  | c | c | c.
FoodⓣPercent by Weight
\^ⓣ_
\^ⓣProteinⓣFatⓣCarbo-
\^ⓣ\^ⓣ\^ⓣhydrate
_
.T&
l | n | n | n.
Applesⓣ.4ⓣ.5ⓣ13.0
Halibutⓣ18.4ⓣ5.2ⓣ. . .
Lima beansⓣ7.5ⓣ.8ⓣ22.0
Milkⓣ3.3ⓣ4.0ⓣ5.0
Mushroomsⓣ3.5ⓣ.4ⓣ6.0
Rye breadⓣ9.0ⓣ.6ⓣ52.7
.TE
```

Output:

Composition of Foods			
Food	Percent by Weight		
	Protein	Fat	Carbo-hydrate
Apples	.4	.5	13.0
Halibut	18.4	5.2	...
Lima beans	7.5	.8	22.0
Milk	3.3	4.0	5.0
Mushrooms	3.5	.4	6.0
Rye bread	9.0	.6	52.7

Input:

```
.TS
allbox;
cfI  s   s
c    cw(1i)   cw(1i)
lp9 lp9 lp9.
New York Area Rocks
EraⓉFormationⓉAge (years)
PrecambrianⓉReading ProngⓉ>1 billion
PaleozoicⓉManhattan ProngⓉ400 million
MesozoicⓉT{
.na
Newark Basin, incl.
Stockton, Lockatong, and Brunswick
formations; also Watchungs
and Palisades.
T}Ⓣ200 million
CenozoicⓉCoastal PlainⓉT{
On Long Island 30,000 years;
Cretaceous sediments redeposited
by recent glaciation.
.ad
T}
.TE
```

Output:

New York Area Rocks		
Era	Formation	Age (years)
Precambrian	Reading Prong	>1 billion
Paleozoic	Manhattan Prong	400 million
Mesozoic	Newark Basin, incl. Stockton, Lockatong, and Brunswick formations; also Watchungs and Palisades.	200 million
Cenozoic	Coastal Plain	On Long Island 30,000 years; Cretaceous sediments redeposited by recent glaciation.

Input:

```
.EQ
delim $$
.EN

. . .

.TS
doublebox;
c c
l l.
NameⓉDefinition
.sp
.vs +2p
GammaⓉ$GAMMA (z) = int sub 0 sup inf  t sup {z-1} e sup -t dt$
SineⓉ$sin (x) = 1 over 2i ( e sup ix - e sup -ix )$
ErrorⓉ$ roman erf (z) = 2 over sqrt pi int sub 0 sup z e sup {-t sup 2} dt$
BesselⓉ$ J sub 0 (z) = 1 over pi int sub 0 sup pi cos ( z sin theta ) d theta $
ZetaⓉ$ zeta (s) = sum from k=1 to inf k sup -s ~~( Re~s > 1)$
.vs -2p
.TE
```

Output:

Name	Definition
Gamma	$\Gamma(z)=\int_0^{\infty} t^{z-1}e^{-t}\,dt$
Sine	$\sin(x)=\frac{1}{2i}(e^{ix}-e^{-ix})$
Error	$\mathrm{erf}(z)=\frac{2}{\sqrt{\pi}}\int_0^{z} e^{-t^2}\,dt$
Bessel	$J_0(z)=\frac{1}{\pi}\int_0^{\pi}\cos(z\sin\theta)\,d\theta$
Zeta	$\zeta(s)=\sum_{k=1}^{\infty}k^{-s}\quad(\mathrm{Re}\ s>1)$

Input:

```
.TS
box, tab(:);
cb s s s s
cp-2 s s s s
c || c | c | c | c
c || c | c | c | c
r2 || n2 | n2 | n2 | n.
Readability of Text
Line Width and Leading for 10-Point Type
=
Line:Set:1-Point:2-Point:4-Point
Width:Solid:Leading:Leading:Leading
_
9 Pica:\-9.3:\-6.0:\-5.3:\-7.1
14 Pica:\-4.5:\-0.6:\-0.3:\-1.7
19 Pica:\-5.0:\-5.1: 0.0:\-2.0
31 Pica:\-3.7:\-3.8:\-2.4:\-3.6
43 Pica:\-9.1:\-9.0:\-5.9:\-8.8
.TE
```

Output:

Readability of Text Line Width and Leading for 10-Point Type				
Line Width	Set Solid	1-Point Leading	2-Point Leading	4-Point Leading
9 Pica	−9.3	−6.0	−5.3	−7.1
14 Pica	−4.5	−0.6	−0.3	−1.7
19 Pica	−5.0	−5.1	0.0	−2.0
31 Pica	−3.7	−3.8	−2.4	−3.6
43 Pica	−9.1	−9.0	−5.9	−8.8

Input:

```
.TS
c s
cip-2 s
l n
a n.
Some London Transport Statistics
(Year 1964)
Railway route milesⓣ244
Tubeⓣ66
Sub-surfaceⓣ22
Surfaceⓣ156
.sp .5
.T&
l r
a r.
Passenger traffic \- railway
Journeysⓣ674 million
Average lengthⓣ4.55 miles
Passenger milesⓣ3,066 million
.T&
l r
a r.
Passenger traffic \- road
Journeysⓣ2,252 million
Average lengthⓣ2.26 miles
Passenger milesⓣ5,094 million
.T&
l n
a n.
.sp .5
Vehiclesⓣ12,521
Railway motor carsⓣ2,905
Railway trailer carsⓣ1,269
Total railwayⓣ4,174
Omnibusesⓣ8,347
.T&
l n
a n.
.sp .5
Staffⓣ73,739
Administrative, etc.ⓣ5,582
Civil engineeringⓣ5,134
Electrical eng.ⓣ1,714
Mech. eng. \- railwayⓣ4,310
Mech. eng. \- roadⓣ9,152
Railway operationsⓣ8,930
Road operationsⓣ35,946
Otherⓣ2,971
.TE
```

Output:

Some London Transport Statistics	
(Year 1964)	
Railway route miles	244
Tube	66
Sub-surface	22
Surface	156
Passenger traffic – railway	
Journeys	674 million
Average length	4.55 miles
Passenger miles	3,066 million
Passenger traffic – road	
Journeys	2,252 million
Average length	2.26 miles
Passenger miles	5,094 million
Vehicles	12,521
Railway motor cars	2,905
Railway trailer cars	1,269
Total railway	4,174
Omnibuses	8,347
Staff	73,739
Administrative, etc.	5,582
Civil engineering	5,134
Electrical eng.	1,714
Mech. eng. – railway	4,310
Mech. eng. – road	9,152
Railway operations	8,930
Road operations	35,946
Other	2,971

Input:

```
.ps 8
.vs 10p
.TS
center box;
c s s
ci s s
c c c
lB l n.
New Jersey Representatives
(Democrats)
.sp .5
NameⓣOffice addressⓣPhone
.sp .5
James J. Florioⓣ23 S. White Horse Pike, Somerdale 08083ⓣ609-627-8222
William J. Hughesⓣ2920 Atlantic Ave., Atlantic City 08401ⓣ609-345-4844
James J. Howardⓣ801 Bangs Ave., Asbury Park 07712ⓣ201-774-1600
Frank Thompson, Jr.ⓣ10 Rutgers Pl., Trenton 08618ⓣ609-599-1619
Andrew Maguireⓣ115 W. Passaic St., Rochelle Park 07662ⓣ201-843-0240
Robert A. RoeⓣU.S.P.O., 194 Ward St., Paterson 07510ⓣ201-523-5152
Henry Helstoskiⓣ666 Paterson Ave., East Rutherford 07073ⓣ201-939-9090
Peter W. Rodino, Jr.ⓣSuite 1435A, 970 Broad St., Newark 07102ⓣ201-645-3213
Joseph G. Minishⓣ308 Main St., Orange 07050ⓣ201-645-6363
Helen S. Meynerⓣ32 Bridge St., Lambertville 08530ⓣ609-397-1830
Dominick V. Danielsⓣ895 Bergen Ave., Jersey City 07306ⓣ201-659-7700
Edward J. PattenⓣNatl. Bank Bldg., Perth Amboy 08861ⓣ201-826-4610
.sp .5
.T&
ci s s
lB l n.
(Republicans)
.sp .5v
Millicent Fenwickⓣ41 N. Bridge St., Somerville 08876ⓣ201-722-8200
Edwin B. Forsytheⓣ301 Mill St., Moorestown 08057ⓣ609-235-6622
Matthew J. Rinaldoⓣ1961 Morris Ave., Union 07083ⓣ201-687-4235
.TE
.ps 10
.vs 12p
```

Output:

New Jersey Representatives *(Democrats)*		
Name	Office address	Phone
James J. Florio	23 S. White Horse Pike, Somerdale 08083	609-627-8222
William J. Hughes	2920 Atlantic Ave., Atlantic City 08401	609-345-4844
James J. Howard	801 Bangs Ave., Asbury Park 07712	201-774-1600
Frank Thompson, Jr.	10 Rutgers Pl., Trenton 08618	609-599-1619
Andrew Maguire	115 W. Passaic St., Rochelle Park 07662	201-843-0240
Robert A. Roe	U.S.P.O., 194 Ward St., Paterson 07510	201-523-5152
Henry Helstoski	666 Paterson Ave., East Rutherford 07073	201-939-9090
Peter W. Rodino, Jr.	Suite 1435A, 970 Broad St., Newark 07102	201-645-3213
Joseph G. Minish	308 Main St., Orange 07050	201-645-6363
Helen S. Meyner	32 Bridge St., Lambertville 08530	609-397-1830
Dominick V. Daniels	895 Bergen Ave., Jersey City 07306	201-659-7700
Edward J. Patten	Natl. Bank Bldg., Perth Amboy 08861	201-826-4610
(Republicans)		
Millicent Fenwick	41 N. Bridge St., Somerville 08876	201-722-8200
Edwin B. Forsythe	301 Mill St., Moorestown 08057	609-235-6622
Matthew J. Rinaldo	1961 Morris Ave., Union 07083	201-687-4235

This is a paragraph of normal text placed here only to indicate where the left and right margins are. In this way the reader can judge the appearance of centered tables or expanded tables, and observe how such tables are formatted.

Input:

```
.TS
expand;
c s s s
c c c c
l l n n.
Bell Labs Locations
Nameⓣ Addressⓣ Area Codeⓣ Phone
Holmdelⓣ Holmdel, N. J. 07733ⓣ 201ⓣ 949-3000
Murray Hillⓣ Murray Hill, N. J. 07974ⓣ 201ⓣ 582-6377
Whippanyⓣ Whippany, N. J. 07981ⓣ 201ⓣ 386-3000
Indian Hillⓣ Naperville, Illinois 60540ⓣ 312ⓣ 690-2000
.TE
```

Output:

Bell Labs Locations			
Name	Address	Area Code	Phone
Holmdel	Holmdel, N. J. 07733	201	949-3000
Murray Hill	Murray Hill, N. J. 07974	201	582-6377
Whippany	Whippany, N. J. 07981	201	386-3000
Indian Hill	Naperville, Illinois 60540	312	690-2000

Input:

```
.TS
box;
cb s s s
c | c | c s
ltiw(1i) | ltw(2i) | lp8 | lw(1.6i)p8.
Some Interesting Places
_
NameⓉDescriptionⓉPractical Information
_
T{
American Museum of Natural History
T}ⓉT{
The collections fill 11.5 acres (Michelin) or 25 acres (MTA)
of exhibition halls on four floors. There is a full-sized replica
of a blue whale and the world's largest star sapphire (stolen in 1964).
T}ⓉHoursⓉ10-5, ex. Sun 11-5, Wed. to 9
\^Ⓣ\^ⓉLocationⓉT{
Central Park West & 79th St.
T}
\^Ⓣ\^ⓉAdmissionⓉDonation: $1.00 asked
\^Ⓣ\^ⓉSubwayⓉAA to 81st St.
\^Ⓣ\^ⓉTelephoneⓉ212-873-4225
_
Bronx ZooⓉT{
About a mile long and .6 mile wide, this is the largest zoo in America.
A lion eats 18 pounds
of meat a day while a sea lion eats 15 pounds of fish.
T}ⓉHoursⓉT{
10-4:30 winter, to 5:00 summer
T}
\^Ⓣ\^ⓉLocationⓉT{
185th St. & Southern Blvd, the Bronx.
T}
\^Ⓣ\^ⓉAdmissionⓉ$1.00, but Tu,We,Th free
\^Ⓣ\^ⓉSubwayⓉ2, 5 to East Tremont Ave.
\^Ⓣ\^ⓉTelephoneⓉ212-933-1759
_
Brooklyn MuseumⓉT{
Five floors of galleries contain American and ancient art.
There are American period rooms and architectural ornaments saved
from wreckers, such as a classical figure from Pennsylvania Station.
T}ⓉHoursⓉWed-Sat, 10-5, Sun 12-5
\^Ⓣ\^ⓉLocationⓉT{
Eastern Parkway & Washington Ave., Brooklyn.
T}
\^Ⓣ\^ⓉAdmissionⓉFree
\^Ⓣ\^ⓉSubwayⓉ2,3 to Eastern Parkway.
\^Ⓣ\^ⓉTelephoneⓉ212-638-5000
_
T{
New-York Historical Society
T}ⓉT{
All the original paintings for Audubon's
.I
Birds of America
.R
are here, as are exhibits of American decorative arts, New York history,
Hudson River school paintings, carriages, and glass paperweights.
T}ⓉHoursⓉT{
Tues-Fri & Sun, 1-5; Sat 10-5
T}
\^Ⓣ\^ⓉLocationⓉT{
Central Park West & 77th St.
T}
\^Ⓣ\^ⓉAdmissionⓉFree
\^Ⓣ\^ⓉSubwayⓉAA to 81st St.
\^Ⓣ\^ⓉTelephoneⓉ212-873-3400
.TE
```

Output:

Some Interesting Places			
Name	Description	Practical Information	
American Museum of Natural History	The collections fill 11.5 acres (Michelin) or 25 acres (MTA) of exhibition halls on four floors. There is a full-sized replica of a blue whale and the world's largest star sapphire (stolen in 1964).	Hours Location Admission Subway Telephone	10-5, ex. Sun 11-5, Wed. to 9 Central Park West & 79th St. Donation: $1.00 asked AA to 81st St. 212-873-4225
Bronx Zoo	About a mile long and .6 mile wide, this is the largest zoo in America. A lion eats 18 pounds of meat a day while a sea lion eats 15 pounds of fish.	Hours Location Admission Subway Telephone	10-4:30 winter, to 5:00 summer 185th St. & Southern Blvd, the Bronx. $1.00, but Tu,We,Th free 2, 5 to East Tremont Ave. 212-933-1759
Brooklyn Museum	Five floors of galleries contain American and ancient art. There are American period rooms and architectural ornaments saved from wreckers, such as a classical figure from Pennsylvania Station.	Hours Location Admission Subway Telephone	Wed-Sat, 10-5, Sun 12-5 Eastern Parkway & Washington Ave., Brooklyn. Free 2,3 to Eastern Parkway. 212-638-5000
New-York Historical Society	All the original paintings for Audubon's *Birds of America* are here, as are exhibits of American decorative arts, New York history, Hudson River school paintings, carriages, and glass paperweights.	Hours Location Admission Subway Telephone	Tues-Fri & Sun, 1-5; Sat 10-5 Central Park West & 77th St. Free AA to 81st St. 212-873-3400

Acknowledgments.

Many thanks are due to J. C. Blinn, who has done a large amount of testing and assisted with the design of the program. He has also written many of the more intelligible sentences in this document and helped edit all of it. All phototypesetting programs on UNIX are dependent on the work of the late J. F. Ossanna, whose assistance with this program in particular had been most helpful. This program is patterned on a table formatter originally written by J. F. Gimpel. The assistance of T. A. Dolotta, B. W. Kernighan, and J. N. Sturman is gratefully acknowledged.

References.

[1] J. F. Ossanna, *NROFF/TROFF User's Manual,* Computing Science Technical Report No. 54, Bell Laboratories, 1976.

[2] K. Thompson and D. M. Ritchie, ''The UNIX Time-Sharing System,'' *Comm. ACM.* **17**, pp. 365–75 (1974).

[3] B. W. Kernighan and L. L. Cherry, ''A System for Typesetting Mathematics,'' *Comm. ACM.* **18**, pp. 151–57 (1975).

[4] M. E. Lesk, *Typing Documents on UNIX,* UNIX Programmer's Manual, Volume 2.

[5] M. E. Lesk and B. W. Kernighan, *Computer Typesetting of Technical Journals on UNIX, Proc. AFIPS NCC*, vol. 46, pp. 879-888 (1977).

[6] J. R. Mashey and D. W. Smith, ''Documentation Tools and Techniques,'' *Proc. 2nd Int. Conf. on Software Engineering,* pp. 177-181 (October, 1976).

List of Tbl Command Characters and Words

<table>
<tr><th>*Command*</th><th>*Meaning*</th><th>*Section*</th></tr>
<tr><td>**a A**</td><td>Alphabetic subcolumn</td><td>2</td></tr>
<tr><td>**allbox**</td><td>Draw box around all items</td><td>1</td></tr>
<tr><td>**b B**</td><td>Boldface item</td><td>2</td></tr>
<tr><td>**box**</td><td>Draw box around table</td><td>1</td></tr>
<tr><td>**c C**</td><td>Centered column</td><td>2</td></tr>
<tr><td>**center**</td><td>Center table in page</td><td>1</td></tr>
<tr><td>**doublebox**</td><td>Doubled box around table</td><td>1</td></tr>
<tr><td>**e E**</td><td>Equal width columns</td><td>2</td></tr>
<tr><td>**expand**</td><td>Make table full line width</td><td>1</td></tr>
<tr><td>**f F**</td><td>Font change</td><td>2</td></tr>
<tr><td>**i I**</td><td>Italic item</td><td>2</td></tr>
<tr><td>**l L**</td><td>Left adjusted column</td><td>2</td></tr>
<tr><td>**n N**</td><td>Numerical column</td><td>2</td></tr>
<tr><td>*nnn*</td><td>Column separation</td><td>2</td></tr>
<tr><td>**p P**</td><td>Point size change</td><td>2</td></tr>
<tr><td>**r R**</td><td>Right adjusted column</td><td>2</td></tr>
<tr><td>**s S**</td><td>Spanned item</td><td>2</td></tr>
<tr><td>**t T**</td><td>Vertical spanning at top</td><td>2</td></tr>
<tr><td>**tab** (*x*)</td><td>Change data separator character</td><td>1</td></tr>
<tr><td>**T{ T}**</td><td>Text block</td><td>3</td></tr>
<tr><td>**v V**</td><td>Vertical spacing change</td><td>2</td></tr>
<tr><td>**w W**</td><td>Minimum width value</td><td>2</td></tr>
<tr><td>**.***xx*</td><td>Included *troff* command</td><td>3</td></tr>
<tr><td>|</td><td>Vertical line</td><td>2</td></tr>
<tr><td>||</td><td>Double vertical line</td><td>2</td></tr>
<tr><td>^</td><td>Vertical span</td><td>2</td></tr>
<tr><td>\^</td><td>Vertical span</td><td>3</td></tr>
<tr><td>**=**</td><td>Double horizontal line</td><td>2,3</td></tr>
<tr><td>_</td><td>Horizontal line</td><td>2,3</td></tr>
<tr><td>**_**</td><td>Short horizontal line</td><td>3</td></tr>
<tr><td>**\R***x*</td><td>Repeat character</td><td>3</td></tr>
</table>

Bell Laboratories
Murray Hill, New Jersey 07974

Computing Science Technical Report No. 69

Some Applications of Inverted Indexes on the UNIX System

M. E. Lesk

June 21, 1978

Some Applications of Inverted Indexes on the UNIX System

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

I. Some Applications of Inverted Indexes – Overview

This memorandum describes a set of programs which make inverted indexes to UNIX* text files, and their application to retrieving and formatting citations for documents prepared using *troff.*

These indexing and searching programs make keyword indexes to volumes of material too large for linear searching. Searches for combinations of single words can be performed quickly. The programs are divided into two phases. The first makes an index from the original data; the second searches the index and retrieves items. Both of these phases are further divided into two parts to separate the data-dependent and algorithm dependent code.

The major current application of these programs is the *troff* preprocessor *refer.* A list of 4300 references is maintained on line, containing primarily papers written and cited by local authors. Whenever one of these references is required in a paper, a few words from the title or author list will retrieve it, and the user need not bother to re-enter the exact citation. Alternatively, authors can use their own lists of papers.

This memorandum is of interest to those who are interested in facilities for searching large but relatively unchanging text files on the UNIX system, and those who are interested in handling bibliographic citations with UNIX *troff.*

II. Updating Publication Lists

This section is a brief note describing the auxiliary programs for managing the updating processing. It is written to aid clerical users in maintaining lists of references. Primarily, the programs described permit a large amount of individual control over the content of publication lists while retaining the usefulness of the files to other users.

III. Manual Pages

This section contains the pages from the UNIX programmer's manual for the *lookall, pubindex,* and *refer* commands. It is useful for reference.

Some Applications of Inverted Indexes on the UNIX System

M. E. Lesk

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction.

The UNIX† system has many utilities (e.g. *grep, awk, lex, egrep, fgrep, ...*) to search through files of text, but most of them are based on a linear scan through the entire file, using some deterministic automaton. This memorandum discusses a program which uses inverted indexes[1] and can thus be used on much larger data bases.

As with any indexing system, of course, there are some disadvantages; once an index is made, the files that have been indexed can not be changed without remaking the index. Thus applications are restricted to those making many searches of relatively stable data. Furthermore, these programs depend on hashing, and can only search for exact matches of whole keywords. It is not possible to look for arithmetic or logical expressions (e.g. ''date greater than 1970'') or for regular expression searching such as that in *lex.*[2]

Currently there are two uses of this software, the *refer* preprocessor to format references, and the *lookall* command to search through all text files on the UNIX system.

The remaining sections of this memorandum discuss the searching programs and their uses. Section 2 explains the operation of the searching algorithm and describes the data collected for use with the *lookall* command. The more important application, *refer* has a user's description in section 3. Section 4 goes into more detail on reference files for the benefit of those who wish to add references to data bases or write new *troff* macros for use with *refer.* The options to make *refer* collect identical citations, or otherwise relocate and adjust references, are described in section 5. The UNIX manual sections for *refer, lookall,* and associated commands are attached as appendices.

2. Searching.

The indexing and searching process is divided into two phases, each made of two parts. These are shown below.

A. Construct the index.

   (1) Find keys — turn the input files into a sequence of tags and keys, where each tag identifies a distinct item in the input and the keys for each such item are the strings under which it is to be indexed.

   (2) Hash and sort — prepare a set of inverted indexes from which, given a set of keys, the appropriate item tags can be found quickly.

B. Retrieve an item in response to a query.

---

†UNIX is a Trademark of Bell Laboratories.

1. D. Knuth, *The Art of Computer Programming: Vol. 3, Sorting and Searching,* Addison-Wesley, Reading, Mass. (1977). See section 6.5.

2. M. E. Lesk, ''Lex — A Lexical Analyzer Generator,'' Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (D).

(3) Search — Given some keys, look through the files prepared by the hashing and sorting facility and derive the appropriate tags.

(4) Deliver — Given the tags, find the original items. This completes the searching process.

The first phase, making the index, is presumably done relatively infrequently. It should, of course, be done whenever the data being indexed change. In contrast, the second phase, retrieving items, is presumably done often, and must be rapid.

An effort is made to separate code which depends on the data being handled from code which depends on the searching procedure. The search algorithm is involved only in steps (2) and (3), while knowledge of the actual data files is needed only by steps (1) and (4). Thus it is easy to adapt to different data files or different search algorithms.

To start with, it is necessary to have some way of selecting or generating keys from input files. For dealing with files that are basically English, we have a key-making program which automatically selects words and passes them to the hashing and sorting program (step 2). The format used has one line for each input item, arranged as follows:

name:start,length (tab) key1 key2 key3 ...

where *name* is the file name, *start* is the starting byte number, and *length* is the number of bytes in the entry.

These lines are the only input used to make the index. The first field (the file name, byte position, and byte count) is the tag of the item and can be used to retrieve it quickly. Normally, an item is either a whole file or a section of a file delimited by blank lines. After the tab, the second field contains the keys. The keys, if selected by the automatic program, are any alphanumeric strings which are not among the 100 most frequent words in English and which are not entirely numeric (except for four-digit numbers beginning 19, which are accepted as dates). Keys are truncated to six characters and converted to lower case. Some selection is needed if the original items are very large. We normally just take the first *n* keys, with *n* less than 100 or so; this replaces any attempt at intelligent selection. One file in our system is a complete English dictionary; it would presumably be retrieved for all queries.

To generate an inverted index to the list of record tags and keys, the keys are hashed and sorted to produce an index. What is wanted, ideally, is a series of lists showing the tags associated with each key. To condense this, what is actually produced is a list showing the tags associated with each hash code, and thus with some set of keys. To speed up access and further save space, a set of three or possibly four files is produced. These files are:

File	Contents
entry	Pointers to posting file for each hash code
posting	Lists of tag pointers for each hash code
tag	Tags for each item
key	Keys for each item (optional)

The posting file comprises the real data: it contains a sequence of lists of items posted under each hash code. To speed up searching, the entry file is an array of pointers into the posting file, one per potential hash code. Furthermore, the items in the lists in the posting file are not referred to by their complete tag, but just by an address in the tag file, which gives the complete tags. The key file is optional and contains a copy of the keys used in the indexing.

The searching process starts with a query, containing several keys. The goal is to obtain all items which were indexed under these keys. The query keys are hashed, and the pointers in the entry file used to access the lists in the posting file. These lists are addresses in the tag file of documents posted under the hash codes derived from the query. The common items from all lists are determined; this must include the items indexed by every key, but may also contain some items which are false drops, since items referenced by the correct hash codes need not actually have contained the correct keys. Normally, if there are several keys in the query, there are not likely to be many false drops in the final combined list even though each hash code is somewhat ambiguous. The actual tags are then obtained from the tag file, and to guard against the possibility that an item has false-dropped on some hash code in the query, the original items are normally obtained from the delivery program (4) and the query keys checked against them by string comparison.

Usually, therefore, the check for bad drops is made against the original file. However, if the key derivation procedure is complex, it may be preferable to check against the keys fed to program (2). In this case the optional key file which contains the keys associated with each item is generated, and the item tag is supplemented by a string

;start,length

which indicates the starting byte number in the key file and the length of the string of keys for each item. This file is not usually necessary with the present key-selection program, since the keys always appear in the original document.

There is also an option (**-C***n*) for coordination level searching. This retrieves items which match all but *n* of the query keys. The items are retrieved in the order of the number of keys that they match. Of course, *n* must be less than the number of query keys (nothing is retrieved unless it matches at least one key).

As an example, consider one set of 4377 references, comprising 660,000 bytes. This included 51,000 keys, of which 5,900 were distinct keys. The hash table is kept full to save space (at the expense of time); 995 of 997 possible hash codes were used. The total set of index files (no key file) included 171,000 bytes, about 26% of the original file size. It took 8 minutes of processor time to hash, sort, and write the index. To search for a single query with the resulting index took 1.9 seconds of processor time, while to find the same paper with a sequential linear search using *grep* (reading all of the tags and keys) took 12.3 seconds of processor time.

We have also used this software to index all of the English stored on our UNIX system. This is the index searched by the *lookall* command. On a typical day there were 29,000 files in our user file system, containing about 152,000,000 bytes. Of these 5,300 files, containing 32,000,000 bytes (about 21%) were English text. The total number of 'words' (determined mechanically) was 5,100,000. Of these 227,000 were selected as keys; 19,000 were distinct, hashing to 4,900 (of 5,000 possible) different hash codes. The resulting inverted file indexes used 845,000 bytes, or about 2.6% of the size of the original files. The particularly small indexes are caused by the fact that keys are taken from only the first 50 non-common words of some very long input files.

Even this large *lookall* index can be searched quickly. For example, to find this document by looking for the keys ''lesk inverted indexes'' required 1.7 seconds of processor time and system time. By comparison, just to search the 800,000 byte dictionary (smaller than even the inverted indexes, let alone the 32,000,000 bytes of text files) with *grep* takes 29 seconds of processor time. The *lookall* program is thus useful when looking for a document which you believe is stored on-line, but do not know where. For example, many memos from the Computing Science Research Center are in its UNIX file system, but it is often difficult to guess where a particular memo might be (it might have several authors, each with many directories, and have been worked on by a secretary with yet more directories). Instructions for the use of the *lookall* command are given in the manual section, shown in the appendix to this memorandum.

The only indexes maintained routinely are those of publication lists and all English files. To make other indexes, the programs for making keys, sorting them, searching the indexes, and delivering answers must be used. Since they are usually invoked as parts of higher-level commands, they are not in the default command directory, but are available to any user in the directory */usr/lib/refer*. Three programs are of interest: *mkey*, which isolates keys from input files; *inv*, which makes an index from a set of keys; and *hunt*, which searches the index and delivers the items. Note that the two parts of the retrieval phase are combined into one program, to avoid the excessive system work and delay which would result from running these as separate processes.

These three commands have a large number of options to adapt to different kinds of input. The user not interested in the detailed description that now follows may skip to section 3, which describes the *refer* program, a packaged-up version of these tools specifically oriented towards formatting references.

Make Keys. The program *mkey* is the key-making program corresponding to step (1) in phase A. Normally, it reads its input from the file names given as arguments, and if there are no arguments it reads from the standard input. It assumes that blank lines in the input delimit separate items, for each of which a different line of keys should be generated. The lines of keys are written on the standard output. Keys are any alphanumeric string in the input not among the most frequent words in English and not entirely numeric (except that all-numeric strings are acceptable if they are between 1900 and 1999). In the output, keys are translated to lower case, and truncated to six characters in length; any associated punctuation is removed. The following flag arguments are recognized by *mkey:*

–c *name*	Name of file of common words; default is */usr/lib/eign.*
–f *name*	Read a list of files from *name* and take each as an input argument.
–i *chars*	Ignore all lines which begin with '%' followed by any character in *chars* .
–k*n*	Use at most *n* keys per input item.
–l*n*	Ignore items shorter than *n* letters long.
–n*m*	Ignore as a key any word in the first *m* words of the list of common English words. The default is 100.
–s	Remove the labels *(file:start,length)* from the output; just give the keys. Used when searching rather than indexing.
–w	Each whole file is a separate item; blank lines in files are irrelevant.

The normal arguments for indexing references are the defaults, which are *–c /usr/lib/eign* , *–n100* , and *–l3* . For searching, the *–s* option is also needed. When the big *lookall* index of all English files is run, the options are *–w* , *–k50* , and *–f (filelist)* . When running on textual input, the *mkey* program processes about 1000 English words per processor second. Unless the *–k* option is used (and the input files are long enough for it to take effect) the output of *mkey* is comparable in size to its input.

Hash and invert. The *inv* program computes the hash codes and writes the inverted files. It reads the output of *mkey* and writes the set of files described earlier in this section. It expects one argument, which is used as the base name for the three (or four) files to be written. Assuming an argument of *Index* (the default) the entry file is named *Index.ia* , the posting file *Index.ib* , the tag file *Index.ic* , and the key file (if present) *Index.id* . The *inv* program recognizes the following options:

–a	Append the new keys to a previous set of inverted files, making new files if there is no old set using the same base name.
–d	Write the optional key file. This is needed when you can not check for false drops by looking for the keys in the original inputs, i.e. when the key derivation procedure is complicated and the output keys are not words from the input files.
–h*n*	The hash table size is *n* (default 997); *n* should be prime. Making *n* bigger saves search time and spends disk space.
–i[u] *name*	Take input from file *name* , instead of the standard input; if **u** is present *name* is unlinked when the sort is started. Using this option permits the sort scratch space to overlap the disk space used for input keys.
–n	Make a completely new set of inverted files, ignoring previous files.
–p	Pipe into the sort program, rather than writing a temporary input file. This saves disk space and spends processor time.
–v	Verbose mode; print a summary of the number of keys which finished indexing.

About half the time used in *inv* is in the contained sort. Assuming the sort is roughly linear, however, a guess at the total timing for *inv* is 250 keys per second. The space used is usually of more importance: the entry file uses four bytes per possible hash (note the **−h** option), and the tag file around 15-20 bytes per item indexed. Roughly, the posting file contains one item for each key instance and one item for each possible hash code; the items are two bytes long if the tag file is less than 65336 bytes long, and the items are four bytes wide if the tag file is greater than 65536 bytes long. To minimize storage, the hash tables should be over-full; for most of the files indexed in this way, there is no other real choice, since the *entry* file must fit in memory.

Searching and Retrieving. The *hunt* program retrieves items from an index. It combines, as mentioned above, the two parts of phase (B): search and delivery. The reason why it is efficient to combine delivery and search is partly to avoid starting unnecessary processes, and partly because the delivery operation must be a part of the search operation in any case. Because of the hashing, the search part takes place in two stages: first items are retrieved which have the right hash codes associated with them, and then the actual items are inspected to determine false drops, i.e. to determine if anything with the right hash codes doesn't really have the right keys. Since the original item is retrieved to check on false drops, it is efficient to present it immediately, rather than only giving the tag as output and later retrieving the item again. If there were a separate key file, this argument would not apply, but separate key files are not common.

Input to *hunt* is taken from the standard input, one query per line. Each query should be in *mkey −s* output format; all lower case, no punctuation. The *hunt* program takes one argument which specifies the base name of the index files to be searched. Only one set of index files can be searched at a time, although many text files may be indexed as a group, of course. If one of the text files has been changed since the index, that file is searched with *fgrep;* this may occasionally slow down the searching, and care should be taken to avoid having many out of date files. The following option arguments are recognized by *hunt:*

−a	Give all output; ignore checking for false drops.
−C*n*	Coordination level *n;* retrieve items with not more than *n* terms of the input missing; default *C0*, implying that each search term must be in the output items.
−F[yn*d***]**	''−Fy'' gives the text of all the items found; ''−Fn'' suppresses them. ''−F*d*'' where *d* is an integer gives the text of the first *d* items. The default is *−Fy.*
−g	Do not use *fgrep* to search files changed since the index was made; print an error comment instead.
−i *string*	Take *string* as input, instead of reading the standard input.
−l *n*	The maximum length of internal lists of candidate items is *n;* default 1000.
−o *string*	Put text output (''−Fy'') in *string;* of use *only* when invoked from another program.
−p	Print hash code frequencies; mostly for use in optimizing hash table sizes.
−T[yn*d***]**	''−Ty'' gives the tags of the items found; ''−Tn'' suppresses them. ''−T*d*'' where *d* is an integer gives the first *d* tags. The default is *−Tn*.
−t *string*	Put tag output (''−Ty'') in *string;* of use *only* when invoked from another program.

The timing of *hunt* is complex. Normally the hash table is overfull, so that there will be many false drops on any single term; but a multi-term query will have few false drops on all terms. Thus if a query is underspecified (one search term) many potential items will be examined and discarded as false drops, wasting time. If the query is overspecified (a dozen search terms) many keys will be examined only to verify that the single item under consideration has that key posted. The variation of search time with number of keys is shown in the table below. Queries of varying length were constructed to retrieve a particular document from the file of references. In the sequence to the left, search terms were chosen so as to select the desired paper as quickly as possible. In the sequence on the right, terms were chosen inefficiently, so that the query did not uniquely select the desired document until four keys had been used. The same document was the target in each case, and the final set of eight keys are also identical; the differences at five, six and seven keys are produced by measurement error, not by the slightly different key lists.

Efficient Keys				Inefficient Keys			
No. keys	Total drops (incl. false)	Retrieved Documents	Search time (seconds)	No. keys	Total drops (incl. false)	Retrieved Documents	Search time (seconds)
1	15	3	1.27	1	68	55	5.96
2	1	1	0.11	2	29	29	2.72
3	1	1	0.14	3	8	8	0.95
4	1	1	0.17	4	1	1	0.18
5	1	1	0.19	5	1	1	0.21
6	1	1	0.23	6	1	1	0.22
7	1	1	0.27	7	1	1	0.26
8	1	1	0.29	8	1	1	0.29

As would be expected, the optimal search is achieved when the query just specifies the answer; however, overspecification is quite cheap. Roughly, the time required by *hunt* can be approximated as 30 milliseconds per search key plus 75 milliseconds per dropped document (whether it is a false drop or a real answer). In general, overspecification can be recommended; it protects the user against additions to the data base which turn previously uniquely-answered queries into ambiguous queries.

The careful reader will have noted an enormous discrepancy between these times and the earlier quoted time of around 1.9 seconds for a search. The times here are purely for the search and retrieval: they are measured by running many searches through a single invocation of the *hunt* program alone. Usually, the UNIX command processor (the shell) must start both the *mkey* and *hunt* processes for each query, and arrange for the output of *mkey* to be fed to the *hunt* program. This adds a fixed overhead of about 1.7 seconds of processor time to any single search. Furthermore, remember that all these times are processor times: on a typical morning on our PDP 11/70 system, with about one dozen people logged on, to obtain 1 second of processor time for the search program took between 2 and 12 seconds of real time, with a median of 3.9 seconds and a mean of 4.8 seconds. Thus, although the work involved in a single search may be only 200 milliseconds, after you add the 1.7 seconds of startup processor time and then assume a 4:1 elapsed/processor time ratio, it will be 8 seconds before any response is printed.

3. Selecting and Formatting References for TROFF

The major application of the retrieval software is *refer,* which is a *troff* preprocessor like *eqn*.[3] It scans its input looking for items of the form

```
.[
imprecise citation
.]
```

where an imprecise citation is merely a string of words found in the relevant bibliographic citation. This is translated into a properly formatted reference. If the imprecise citation does not correctly identify a single paper (either selecting no papers or too many) a message is given. The data base of citations searched may be tailored to each system, and individual users may specify their own citation files. On our system, the default data base is accumulated from the publication lists of the members of our organization, plus about half a dozen personal bibliographies that were collected. The present total is about 4300 citations, but this increases steadily. Even now, the data base covers a large fraction of local citations.

For example, the reference for the *eqn* paper above was specified as

3. B. W. Kernighan and L. L. Cherry, ''A System for Typesetting Mathematics,'' *Comm. Assoc. Comp. Mach.* **18**, pp.151-157 (March 1975).

```
...
preprocessor like
.I eqn.
.[
kernighan cherry acm 1975
.]
It scans its input looking for items
...
```

This paper was itself printed using *refer.* The above input text was processed by *refer* as well as *tbl* and *troff* by the command

refer memo-file | *tbl* | *troff* −*ms*

and the reference was automatically translated into a correct citation to the ACM paper on mathematical typesetting.

The procedure to use to place a reference in a paper using *refer* is as follows. First, use the *lookbib* command to check that the paper is in the data base and to find out what keys are necessary to retrieve it. This is done by typing *lookbib* and then typing some potential queries until a suitable query is found. For example, had one started to find the *eqn* paper shown above by presenting the query

```
$ lookbib
kernighan cherry
(EOT)
```

lookbib would have found several items; experimentation would quickly have shown that the query given above is adequate. Overspecifying the query is of course harmless; it is even desirable, since it decreases the risk that a document added to the publication data base in the future will be retrieved in addition to the intended document. The extra time taken by even a grossly overspecified query is quite small. A particularly careful reader may have noticed that ‘‘acm’’ does not appear in the printed citation; we have supplemented some of the data base items with extra keywords, such as common abbreviations for journals or other sources, to aid in searching.

If the reference is in the data base, the query that retrieved it can be inserted in the text, between **.[** and **.]** brackets. If it is not in the data base, it can be typed into a private file of references, using the format discussed in the next section, and then the −**p** option used to search this private file. Such a command might read (if the private references are called *myfile*)

refer −*p myfile document* | *tbl* | *eqn* | *troff* −*ms* . . .

where *tbl* and/or *eqn* could be omitted if not needed. The use of the −*ms* macros[4] or some other macro package, however, is essential. *Refer* only generates the data for the references; exact formatting is done by some macro package, and if none is supplied the references will not be printed.

By default, the references are numbered sequentially, and the −*ms* macros format references as footnotes at the bottom of the page. This memorandum is an example of that style. Other possibilities are discussed in section 5 below.

4. Reference Files.

A reference file is a set of bibliographic references usable with *refer.* It can be indexed using the software described in section 2 for fast searching. What *refer* does is to read the input document stream, looking for imprecise citation references. It then searches through reference files to find the full citations, and inserts them into the document. The format of the full citation is arranged to make it convenient for a macro package, such as the −*ms* macros, to format the reference for printing. Since the format of the final reference is determined by the desired style of output, which is determined by the

4. M. E. Lesk, *Typing Documents on UNIX and GCOS: The -ms Macros for Troff*, 1977.

macros used, *refer* avoids forcing any kind of reference appearance. All it does is define a set of string registers which contain the basic information about the reference; and provide a macro call which is expanded by the macro package to format the reference. It is the responsibility of the final macro package to see that the reference is actually printed; if no macros are used, and the output of *refer* fed untranslated to *troff,* nothing at all will be printed.

The strings defined by *refer* are taken directly from the files of references, which are in the following format. The references should be separated by blank lines. Each reference is a sequence of lines beginning with **%** and followed by a key-letter. The remainder of that line, and successive lines until the next line beginning with **%**, contain the information specified by the key-letter. In general, *refer* does not interpret the information, but merely presents it to the macro package for final formatting. A user with a separate macro package, for example, can add new key-letters or use the existing ones for other purposes without bothering *refer.*

The meaning of the key-letters given below, in particular, is that assigned by the −*ms* macros. Not all information, obviously, is used with each citation. For example, if a document is both an internal memorandum and a journal article, the macros ignore the memorandum version and cite only the journal article. Some kinds of information are not used at all in printing the reference; if a user does not like finding references by specifying title or author keywords, and prefers to add specific keywords to the citation, a field is available which is searched but not printed (**K**).

The key letters currently recognized by *refer* and −*ms,* with the kind of information implied, are:

Key	Information specified	Key	Information specified
A	Author's name	N	Issue number
B	Title of book containing item	O	Other information
C	City of publication	P	Page(s) of article
D	Date	R	Technical report reference
E	Editor of book containing item	T	Title
G	Government (NTIS) ordering number	V	Volume number
I	Issuer (publisher)		
J	Journal name		
K	Keys (for searching)	X	or
L	Label	Y	or
M	Memorandum label	Z	Information not used by *refer*

For example, a sample reference could be typed as:

```
%T Bounds on the Complexity of the Maximal
Common Subsequence Problem
%Z ctr127
%A A. V. Aho
%A D. S. Hirschberg
%A J. D. Ullman
%J J. ACM
%V 23
%N 1
%P 1-12
%M abcd-78
%D Jan. 1976
```

Order is irrelevant, except that authors are shown in the order given. The output of *refer* is a stream of string definitions, one for each of the fields of each reference, as shown below.

```
.]-
.ds [A authors' names ...
.ds [T title ...
.ds [J journal ...
...
.] [ type-number
```

The *refer* program, in general, does not concern itself with the significance of the strings. The different fields are treated identically by *refer*, except that the X, Y and Z fields are ignored (see the **−i** option of *mkey*) in indexing and searching. All *refer* does is select the appropriate citation, based on the keys. The macro package must arrange the strings so as to produce an appropriately formatted citation. In this process, it uses the convention that the 'T' field is the title, the 'J' field the journal, and so forth.

The *refer* program does arrange the citation to simplify the macro package's job, however. The special macro **.]–** precedes the string definitions and the special macro **.] [** follows. These are changed from the input **.[** and **.]** so that running the same file through *refer* again is harmless. The **.]–** macro can be used by the macro package to initialize. The **.] [** macro, which should be used to print the reference, is given an argument *type-number* to indicate the kind of reference, as follows:

Value	Kind of reference
1	Journal article
2	Book
3	Article within book
4	Technical report
5	Bell Labs technical memorandum
0	Other

The type is determined by the presence or absence of particular fields in the citation (a journal article must have a 'J' field, a book must have an 'I' field, and so forth). To a small extent, this violates the above rule that *refer* does not concern itself with the contents of the citation; however, the classification of the citation in *troff* macros would require a relatively expensive and obscure program. Any macro writer may, of course, preserve consistency by ignoring the argument to the **.] [** macro.

The reference is flagged in the text with the sequence

```
\*([.number\*(.]
```

where *number* is the footnote number. The strings **[.** and **.]** should be used by the macro package to format the reference flag in the text. These strings can be replaced for a particular footnote, as described in section 5. The footnote number (or other signal) is available to the reference macro **.] [** as the string register **[F**. To simplify dealing with a text reference that occurs at the end of a sentence, *refer* treats a reference which follows a period in a special way. The period is removed, and the reference is preceded by a call for the string **<.** and followed by a call for the string **>.** For example, if a reference follows ''end.'' it will appear as

```
end\*(<.\*([.number\*(.]\*(>.
```

where *number* is the footnote number. The macro package should turn either the string **>.** or **<.** into a period and delete the other one. This permits the output to have either the form ''end[31].'' or ''end.[31]'' as the macro package wishes. Note that in one case the period precedes the number and in the other it follows the number.

In some cases users wish to suspend the searching, and merely use the reference macro formatting. That is, the user doesn't want to provide a search key between **.[** and **.]** brackets, but merely the reference lines for the appropriate document. Alternatively, the user can wish to add a few fields to those in the reference as in the standard file, or override some fields. Altering or replacing fields, or supplying whole references, is easily done by inserting lines beginning with **%**; any such line is taken as direct input to the reference processor rather than keys to be searched. Thus

```
.[
key1 key2 key3 ...
%Q New format item
%R Override report name
.]
```

makes the indicates changes to the result of searching for the keys. All of the search keys must be given before the first **%** line.

If no search keys are provided, an entire citation can be provided in-line in the text. For example, if the *eqn* paper citation were to be inserted in this way, rather than by searching for it in the data base, the input would read

```
...
preprocessor like
.I eqn.
.[
%A B. W. Kernighan
%A L. L. Cherry
%T A System for Typesetting Mathematics
%J Comm. ACM
%V 18
%N 3
%P 151-157
%D March 1975
.]
It scans its input looking for items
...
```

This would produce a citation of the same appearance as that resulting from the file search.

As shown, fields are normally turned into *troff* strings. Sometimes users would rather have them defined as macros, so that other *troff* commands can be placed into the data. When this is necessary, simply double the control character **%** in the data. Thus the input

```
.[
%V 23
%%M
Bell Laboratories,
Murray Hill, N.J. 07974
.]
```

is processed by *refer* into

```
.ds [V 23
.de [M
Bell Laboratories,
Murray Hill, N.J. 07974
..
```

The information after **%%M** is defined as a macro to be invoked by **.[M** while the information after **%V** is turned into a string to be invoked by *(**[V**. At present *−ms* expects all information as strings.

5. Collecting References and other Refer Options

Normally, the combination of *refer* and *−ms* formats output as *troff* footnotes which are consecutively numbered and placed at the bottom of the page. However, options exist to place the references at the end; to arrange references alphabetically by senior author; and to indicate references by strings in the text of the form [Name1975a] rather than by number. Whenever references are not placed at the bottom of a page identical references are coalesced.

For example, the −**e** option to *refer* specifies that references are to be collected; in this case they are output whenever the sequence

```
.[
$LIST$
.]
```

is encountered. Thus, to place references at the end of a paper, the user would run *refer* with the −*e* option and place the above $LIST$ commands after the last line of the text. *Refer* will then move all the references to that point. To aid in formatting the collected references, *refer* writes the references preceded by the line

```
.]<
```

and followed by the line

```
.]>
```

to invoke special macros before and after the references.

Another possible option to *refer* is the −**s** option to specify sorting of references. The default, of course, is to list references in the order presented. The −**s** option implies the −**e** option, and thus requires a

```
.[
$LIST$
.]
```

entry to call out the reference list. The −**s** option may be followed by a string of letters, numbers, and '+' signs indicating how the references are to be sorted. The sort is done using the fields whose key-letters are in the string as sorting keys; the numbers indicate how many of the fields are to be considered, with '+' taken as a large number. Thus the default is −**sAD** meaning ''Sort on senior author, then date.'' To sort on all authors and then title, specify −**sA+T**. And to sort on two authors and then the journal, write −**sA2J**.

Other options to *refer* change the signal or label inserted in the text for each reference. Normally these are just sequential numbers, and their exact placement (within brackets, as superscripts, etc.) is determined by the macro package. The −**l** option replaces reference numbers by strings composed of the senior author's last name, the date, and a disambiguating letter. If a number follows the **l** as in −**l3** only that many letters of the last name are used in the label string. To abbreviate the date as well the form **-l***m,n* shortens the last name to the first *m* letters and the date to the last *n* digits. For example, the option −**l3,2** would refer to the *eqn* paper (reference 3) by the signal *Ker75a*, since it is the first cited reference by Kernighan in 1975.

A user wishing to specify particular labels for a private bibliography may use the −**k** option. Specifying −**k***x* causes the field *x* to be used as a label. The default is **L**. If this field ends in −, that character is replaced by a sequence letter; otherwise the field is used exactly as given.

If none of the *refer*-produced signals are desired, the −**b** option entirely suppresses automatic text signals.

If the user wishes to override the −*ms* treatment of the reference signal (which is normally to enclose the number in brackets in *nroff* and make it a superscript in *troff*) this can be done easily. If the lines **.[** or **.]** contain anything following these characters, the remainders of these lines are used to surround the reference signal, instead of the default. Thus, for example, to say ''See reference (2).'' and avoid ''See reference.[2]'' the input might appear

```
See reference
.[ (
imprecise citation ...
.]).
```

Note that blanks are significant in this construction. If a permanent change is desired in the style of reference signals, however, it is probably easier to redefine the strings **[.** and **.]** (which are used to bracket each signal) than to change each citation.

Although normally *refer* limits itself to retrieving the data for the reference, and leaves to a macro package the job of arranging that data as required by the local format, there are two special options for rearrangements that can not be done by macro packages. The **−c** option puts fields into all upper case (CAPS-SMALL CAPS in *troff* output). The key-letters indicated what information is to be translated to upper case follow the **c**, so that **−cAJ** means that authors' names and journals are to be in caps. The **−a** option writes the names of authors last name first, that is *A. D. Hall, Jr.* is written as *Hall, A. D. Jr*. The citation form of the *Journal of the ACM*, for example, would require both **−cA** and **−a** options. This produces authors' names in the style *KERNIGHAN, B. W. AND CHERRY, L. L.* for the previous example. The **−a** option may be followed by a number to indicate how many author names should be reversed; **−a1** (without any **−c** option) would produce *Kernighan, B. W. and L. L. Cherry,* for example.

Finally, there is also the previously-mentioned **−p** option to let the user specify a private file of references to be searched before the public files. Note that *refer* does not insist on a previously made index for these files. If a file is named which contains reference data but is not indexed, it will be searched (more slowly) by *refer* using *fgrep.* In this way it is easy for users to keep small files of new references, which can later be added to the public data bases.

NROFF/TROFF User's Manual

Joseph F. Ossanna

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

NROFF and TROFF are text processors under the PDP-11 UNIX Time-Sharing System[1] that format text for typewriter-like terminals and for a Graphic Systems phototypesetter, respectively. They accept lines of text interspersed with lines of format control information and format the text into a printable, paginated document having a user-designed style. NROFF and TROFF offer unusual freedom in document styling, including: arbitrary style headers and footers; arbitrary style footnotes; multiple automatic sequence numbering for paragraphs, sections, etc; multiple column output; dynamic font and point-size control; arbitrary horizontal and vertical local motions at any point; and a family of automatic overstriking, bracket construction, and line drawing functions.

NROFF and TROFF are highly compatible with each other and it is almost always possible to prepare input acceptable to both. Conditional input is provided that enables the user to embed input expressly destined for either program. NROFF can prepare output directly for a variety of terminal types and is capable of utilizing the full resolution of each terminal.

Usage

The general form of invoking NROFF (or TROFF) at UNIX command level is

nroff *options files* (or **troff** *options files*)

where *options* represents any of a number of option arguments and *files* represents the list of files containing the document to be formatted. An argument consisting of a single minus (−) is taken to be a file name corresponding to the standard input. If no file names are given input is taken from the standard input. The options, which may appear in any order so long as they appear before the files, are:

Option	*Effect*
−**o***list*	Print only pages whose page numbers appear in *list*, which consists of comma-separated numbers and number ranges. A number range has the form *N*−*M* and means pages *N* through *M;* a initial −*N* means from the beginning to page *N;* and a final *N*− means from *N* to the end.
−**n***N*	Number first generated page *N*.
−**s***N*	Stop every *N* pages. NROFF will halt prior to every *N* pages (default *N*=1) to allow paper loading or changing, and will resume upon receipt of a newline. TROFF will stop the phototypesetter every *N* pages, produce a trailer to allow changing cassettes, and will resume after the phototypesetter START button is pressed.
−**m***name*	Prepends the macro file **/usr/lib/tmac.***name* to the input *files*.
−**r***aN*	Register *a* (one-character) is set to *N*.
−**i**	Read standard input after the input files are exhausted.
−**q**	Invoke the simultaneous input-output mode of the **rd** request.

NROFF Only

−T*name* Specifies the name of the output terminal type. Currently defined names are **37** for the (default) Model 37 Teletype®, **tn300** for the GE TermiNet 300 (or any terminal without half-line capabilities), **300S** for the DASI-300S, **300** for the DASI-300, and **450** for the DASI-450 (Diablo Hyterm).

−e Produce equally-spaced words in adjusted lines, using full terminal resolution.

TROFF Only

−t Direct output to the standard output instead of the phototypesetter.

−f Refrain from feeding out paper and stopping phototypesetter at the end of the run.

−w Wait until phototypesetter is available, if currently busy.

−b TROFF will report whether the phototypesetter is busy or available. No text processing is done.

−a Send a printable (ASCII) approximation of the results to the standard output.

−p*N* Print all characters in point size *N* while retaining all prescribed spacings and motions, to reduce phototypesetter elasped time.

−g Prepare output for the Murray Hill Computation Center phototypesetter and direct it to the standard output.

Each option is invoked as a separate argument; for example,

nroff **−o***4,8–10* **−T***300S* **−m***abc* *file1* *file2*

requests formatting of pages 4, 8, 9, and 10 of a document contained in the files named *file1* and *file2*, specifies the output terminal as a DASI-300S, and invokes the macro package *abc*.

Various pre- and post-processors are available for use with NROFF and TROFF. These include the equation preprocessors NEQN and EQN[2] (for NROFF and TROFF respectively), and the table-construction preprocessor TBL[3]. A reverse-line postprocessor COL[4] is available for multiple-column NROFF output on terminals without reverse-line ability; COL expects the Model 37 Teletype escape sequences that NROFF produces by default. TK[4] is a 37 Teletype simulator postprocessor for printing NROFF output on a Tektronix 4014. TCAT[4] is phototypesetter-simulator postprocessor for TROFF that produces an approximation of phototypesetter output on a Tektronix 4014. For example, in

tbl *files* | **eqn** | **troff** **−t** *options* | **tcat**

the first | indicates the piping of TBL's output to EQN's input; the second the piping of EQN's output to TROFF's input; and the third indicates the piping of TROFF's output to TCAT. GCAT[4] can be used to send TROFF (**−g**) output to the Murray Hill Computation Center.

The remainder of this manual consists of: a Summary and Index; a Reference Manual keyed to the index; and a set of Tutorial Examples. Another tutorial is [5].

Joseph F. Ossanna

References

[1] K. Thompson, D. M. Ritchie, *UNIX Programmer's Manual,* Sixth Edition (May 1975).

[2] B. W. Kernighan, L. L. Cherry, *Typesetting Mathematics — User's Guide (Second Edition),* Bell Laboratories internal memorandum.

[3] M. E. Lesk, *Tbl — A Program to Format Tables,* Bell Laboratories internal memorandum.

[4] Internal on-line documentation, on UNIX.

[5] B. W. Kernighan, *A TROFF Tutorial*, Bell Laboratories internal memorandum.

SUMMARY AND INDEX

Request Form	*Initial Value**	*If No Argument*	*Notes#*	*Explanation*
1. General Explanation				
2. Font and Character Size Control				
.ps ±*N*	10 point	previous	E	Point size; also **\s**±*N*.†
.ss *N*	12⁄36 em	ignored	E	Space-character size set to *N*⁄36 em.†
.cs *F N M*	off	-	P	Constant character space (width) mode (font *F*).†
.bd *F N*	off	-	P	Embolden font *F* by *N*−1 units.†
.bd S *F N*	off	-	P	Embolden Special Font when current font is *F*.†
.ft *F*	Roman	previous	E	Change to font *F* = *x*, *xx*, or 1-4. Also **\f***x*, **\f(***xx*, **\f***N*.
.fp *N F*	R,I,B,S	ignored	-	Font named *F* mounted on physical position 1≤*N*≤4.
3. Page Control				
.pl ±*N*	11 in	11 in	**v**	Page length.
.bp ±*N*	*N*=1	-	B‡,**v**	Eject current page; next page number *N*.
.pn ±*N*	*N*=1	ignored	-	Next page number *N*.
.po ±*N*	0; 26⁄27 in	previous	**v**	Page offset.
.ne *N*	-	*N*=1*V*	D,**v**	Need *N* vertical space (*V* = vertical spacing).
.mk *R*	none	internal	D	Mark current vertical place in register *R*.
.rt ±*N*	none	internal	D,**v**	Return *(upward only)* to marked vertical place.
4. Text Filling, Adjusting, and Centering				
.br	-	-	B	Break.
.fi	fill	-	B,E	Fill output lines.
.nf	fill	-	B,E	No filling or adjusting of output lines.
.ad *c*	adj,both	adjust	E	Adjust output lines with mode *c*.
.na	adjust	-	E	No output line adjusting.
.ce *N*	off	*N*=1	B,E	Center following *N* input text lines.
5. Vertical Spacing				
.vs *N*	1⁄6in;12pts	previous	E,**p**	Vertical base line spacing (*V*).
.ls *N*	*N*=1	previous	E	Output *N*−1 *V*s after each text output line.
.sp *N*	-	*N*=1*V*	B,**v**	Space vertical distance *N in either direction*.
.sv *N*	-	*N*=1*V*	**v**	Save vertical distance *N*.
.os	-	-	-	Output saved vertical distance.
.ns	space	-	D	Turn no-space mode on.
.rs	-	-	D	Restore spacing; turn no-space mode off.
6. Line Length and Indenting				
.ll ±*N*	6.5 in	previous	E,**m**	Line length.
.in ±*N*	*N*=0	previous	B,E,**m**	Indent.
.ti ±*N*	-	ignored	B,E,**m**	Temporary indent.
7. Macros, Strings, Diversion, and Position Traps				
.de *xx yy*	-	*.yy*=**..**	-	Define or redefine macro *xx;* end at call of *yy*.
.am *xx yy*	-	*.yy*=**..**	-	Append to a macro.
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string*.

---

*Values separated by "**;**" are for NROFF and TROFF respectively.

#Notes are explained at the end of this Summary and Index

†No effect in NROFF.

‡The use of " ´ " as control character (instead of "**.**") suppresses the break function.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.as *xx string*	-	ignored	-	Append *string* to string *xx*.
.rm *xx*	-	ignored	-	Remove request, macro, or string.
.rn *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*.
.di *xx*	-	end	D	Divert output to macro *xx*.
.da *xx*	-	end	D	Divert and append to *xx*.
.wh *N xx*	-	-	**v**	Set location trap; negative is w.r.t. page bottom.
.ch *xx N*	-	-	**v**	Change trap location.
.dt *N xx*	-	off	D,**v**	Set a diversion trap.
.it *N xx*	-	off	E	Set an input-line count trap.
.em *xx*	none	none	-	End macro is *xx*.

8. Number Registers

.nr *R* ±*N M*	-	**u**	Define and set number register *R*; auto-increment by *M*.	
.af *R c*	arabic	-	-	Assign format to register *R* (*c*=**1**, **i**, **I**, **a**, **A**).
.rr *R*	-	-	-	Remove register *R*.

9. Tabs, Leaders, and Fields

.ta *Nt* ...	0.8; 0.5in	none	E,**m**	Tab settings; *left* type, unless *t*=**R**(right), **C**(centered).
.tc *c*	none	none	E	Tab repetition character.
.lc *c*	**.**	none	E	Leader repetition character.
.fc *a b*	off	off	-	Set field delimiter *a* and pad character *b*.

10. Input and Output Conventions and Character Translations

.ec *c*	\	\	-	Set escape character.
.eo	on	-	-	Turn off escape character mechanism.
.lg *N*	-; on	on	-	Ligature mode on if *N*>0.
.ul *N*	off	*N*=1	E	Underline (italicize in TROFF) *N* input lines.
.cu *N*	off	*N*=1	E	Continuous underline in NROFF; like **ul** in TROFF.
.uf *F*	Italic	Italic	-	Underline font set to *F* (to be switched to by **ul**).
.cc *c*	**.**	**.**	E	Set control character to *c*.
.c2 *c*	´	´	E	Set nobreak control character to *c*.
.tr *abcd....*	none	-	O	Translate *a* to *b*, etc. on output.

11. Local Horizontal and Vertical Motions, and the Width Function

12. Overstrike, Bracket, Line-drawing, and Zero-width Functions

13. Hyphenation.

.nh	hyphenate	-	E	No hyphenation.
.hy *N*	hyphenate	hyphenate	E	Hyphenate; *N* = mode.
.hc *c*	\%	\%	E	Hyphenation indicator character *c*.
.hw *word1* ...	ignored	-	Exception words.	

14. Three Part Titles.

.tl *´left´center´right´*		-	-	Three part title.
.pc *c*	%	off	-	Page number character.
.lt ±*N*	6.5 in	previous	E,**m**	Length of title.

15. Output Line Numbering.

.nm ±*N M S I*		off	E	Number mode on or off, set parameters.
.nn *N*	-	*N*=1	E	Do not number next *N* lines.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*

16. Conditional Acceptance of Input

.if *c anything*		-	-	If condition *c* true, accept *anything* as input, for multi-line use \{*anything*\}.
.if !*c anything*		-	-	If condition *c* false, accept *anything*.
.if *N anything*-		**u**	If expression $N > 0$, accept *anything*.	
.if !*N anything*		-	**u**	If expression $N \leq 0$, accept *anything*.
.if *´string1´string2´ anything*			-	If *string1* identical to *string2*, accept *anything*.
.if !*´string1´string2´ anything*			-	If *string1* not identical to *string2*, accept *anything*.
.ie *c anything* -		**u**	If portion of if-else; all above forms (like **if**).	
.el *anything*		-	-	Else portion of if-else.

17. Environment Switching.

.ev *N*	*N*=0	previous	-	Environment switched (*push down*).

18. Insertions from the Standard Input

.rd *prompt*	-	*prompt*=BEL	-	Read insertion.
.ex	-	-	-	Exit from NROFF/TROFF.

19. Input/Output File Switching

.so *filename*		-	-	Switch source file *(push down)*.
.nx *filename*		end-of-file	-	Next file.
.pi *program*		-	-	Pipe output to *program* (NROFF only).

20. Miscellaneous

.mc *c N*	-	off	E,**m**	Set margin character *c* and separation *N*.
.tm *string*	-	newline	-	Print *string* on terminal (UNIX standard message output).
.ig *yy*	-	*.yy*=**..**	-	Ignore till call of *yy*.
.pm *t*	-	all	-	Print macro names and sizes; if *t* present, print only total of sizes.
.fl	-	-	B	Flush output buffer.

21. Output and Error Messages

Notes-

B	Request normally causes a break.
D	Mode or relevant parameters associated with current diversion level.
E	Relevant parameters are a part of the current environment.
O	Must stay in effect until logical output.
P	Mode must be still or again in effect at the time of physical output.
v,**p**,**m**,**u**	Default scale indicator; if not specified, scale indicators are *ignored*.

Alphabetical Request and Section Number Cross Reference

ad 4	cc 10	ds 7	fc 9	ie 16	ll 6	nh 13	pi 19	rn 7	ta 9	vs 5
af 8	ce 4	dt 7	fi 4	if 16	ls 5	nm 15	pl 3	rr 8	tc 9	wh 7
am 7	ch 7	ec 10	fl 20	ig 20	lt 14	nn 15	pm 20	rs 5	ti 6	
as 7	cs 2	el 16	fp 2	in 6	mc 20	nr 8	pn 3	rt 3	tl 14	
bd 2	cu 10	em 7	ft 2	it 7	mk 3	ns 5	po 3	so 19	tm 20	
bp 3	da 7	eo 10	hc 13	lc 9	na 4	nx 19	ps 2	sp 5	tr 10	
br 4	de 7	ev 17	hw 13	lg 10	ne 3	os 5	rd 18	ss 2	uf 10	
c2 10	di 7	ex 18	hy 13	li 10	nf 4	pc 14	rm 7	sv 5	ul 10	

Escape Sequences for Characters, Indicators, and Functions

Section Reference	*Escape Sequence*	*Meaning*
10.1	\\	\ (to prevent or delay the interpretation of \ )
10.1	**\e**	Printable version of the *current* escape character.
2.1	\´	´ (acute accent); equivalent to **\(aa**
2.1	\`	` (grave accent); equivalent to **\(ga**
2.1	\−	− Minus sign in the *current* font
7	**\.**	Period (dot) (see **de**)
11.1	\(space)	Unpaddable space-size space character
11.1	**\0**	Digit width space
11.1	\\|	1⁄6 em narrow space character (zero width in NROFF)
11.1	\^	1⁄12 em half-narrow space character (zero width in NROFF)
4.1	**\&**	Non-printing, zero width character
10.6	**\!**	Transparent line indicator
10.7	\"	Beginning of comment
7.3	**\$***N*	Interpolate argument 1≤*N*≤9
13	**\%**	Default optional hyphenation character
2.1	\(*xx*	Character named *xx*
7.1	*_x_, *(_xx_	Interpolate string *x* or *xx*
9.1	**\a**	Non-interpreted leader character
12.3	**\b**´*abc...*´	Bracket building function
4.2	**\c**	Interrupt text processing
11.1	**\d**	Forward (down) 1⁄2 em vertical motion (1⁄2 line in NROFF)
2.2	**\f***x*,**\f(***xx*,**\f***N*	Change to font named *x* or *xx*, or position *N*
11.1	**\h**´*N* ´	Local horizontal motion; move right *N (negative left)*
11.3	**\k***x*	Mark horizontal *input* place in register *x*
12.4	**\l** ´*Nc*´	Horizontal line drawing function (optionally with *c*)
12.4	**\L**´*Nc*´	Vertical line drawing function (optionally with *c*)
8	**\n***x*,**\n(***xx*	Interpolate number register *x* or *xx*
12.1	**\o**´*abc...*´	Overstrike characters *a, b, c, ...*
4.1	**\p**	Break and spread output line
11.1	**\r**	Reverse 1 em vertical motion (reverse line in NROFF)
2.3	**\s***N*, **\s**±*N*	Point-size change function
9.1	**\t**	Non-interpreted horizontal tab
11.1	**\u**	Reverse (up) 1⁄2 em vertical motion (1⁄2 line in NROFF)
11.1	**\v**´*N* ´	Local vertical motion; move down *N (negative up)*
11.2	**\w**´*string*´	Interpolate width of *string*
5.2	**\x**´*N* ´	Extra line-space function *(negative before, positive after)*
12.2	**\z***c*	Print *c* with zero width (without spacing)
16	\{	Begin conditional input
16	\}	End conditional input
10.7	\(newline)	Concealed (ignored) newline
-	*X*	*X*, any character *not* listed above

The escape sequences \\, **\.**, \", **\$**, *, **\a**, **\n**, **\t**, and \(newline) are interpreted in *copy mode* (§7.2).

Predefined General Number Registers

Section Reference	*Register Name*	*Description*
3	**%**	Current page number.
11.2	**ct**	Character type (set by *width* function).
7.4	**dl**	Width (maximum) of last completed diversion.
7.4	**dn**	Height (vertical size) of last completed diversion.
-	**dw**	Current day of the week (1-7).
-	**dy**	Current day of the month (1-31).
11.3	**hp**	Current horizontal place on *input* line.
15	**ln**	Output line number.
-	**mo**	Current month (1-12).
4.1	**nl**	Vertical position of last printed text base-line.
11.2	**sb**	Depth of string below base line (generated by *width* function).
11.2	**st**	Height of string above base line (generated by *width* function).
-	**yr**	Last two digits of current year.

Predefined Read-Only Number Registers

Section Reference	*Register Name*	*Description*
7.3	**.$**	Number of arguments available at the current macro level.
-	**.A**	Set to 1 in TROFF, if −**a** option used; always 1 in NROFF.
11.1	**.H**	Available horizontal resolution in basic units.
-	**.T**	Set to 1 in NROFF, if −**T** option used; always 0 in TROFF.
11.1	**.V**	Available vertical resolution in basic units.
5.2	**.a**	Post-line extra line-space most recently utilized using **\x**´*N*´.
-	**.c**	Number of *lines* read from current input file.
7.4	**.d**	Current vertical place in current diversion; equal to **nl**, if no diversion.
2.2	**.f**	Current font as physical quadrant (1-4).
4	**.h**	Text base-line high-water mark on current page or diversion.
6	**.i**	Current indent.
6	**.l**	Current line length.
4	**.n**	Length of text portion on previous output line.
3	**.o**	Current page offset.
3	**.p**	Current page length.
2.3	**.s**	Current point size.
7.5	**.t**	Distance to the next trap.
4.1	**.u**	Equal to 1 in fill mode and 0 in nofill mode.
5.1	**.v**	Current vertical line spacing.
11.2	**.w**	Width of previous character.
-	**.x**	Reserved version-dependent register.
-	**.y**	Reserved version-dependent register.
7.4	**.z**	Name of current diversion.

REFERENCE MANUAL

1. General Explanation

1.1. Form of input. Input consists of *text lines*, which are destined to be printed, interspersed with *control lines*, which set parameters or otherwise control subsequent processing. Control lines begin with a *control character*—normally **.** (period) or ´ (acute accent)—followed by a one or two character name that specifies a basic *request* or the substitution of a user-defined *macro* in place of the control line. The control character ´ suppresses the *break* function—the forced output of a partially filled line—caused by certain requests. The control character may be separated from the request/macro name by white space (spaces and/or tabs) for esthetic reasons. Names must be followed by either space or newline. Control lines with unrecognized names are ignored.

Various special functions may be introduced anywhere in the input by means of an *escape* character, normally \. For example, the function **\n***R* causes the interpolation of the contents of the *number register R* in place of the function; here *R* is either a single character name as in **\n***x*, or left-parenthesis-introduced, two-character name as in **\n(***xx*.

1.2. Formatter and device resolution. TROFF internally uses 432 units/inch, corresponding to the Graphic Systems phototypesetter which has a horizontal resolution of 1/432 inch and a vertical resolution of 1/144 inch. NROFF internally uses 240 units/inch, corresponding to the least common multiple of the horizontal and vertical resolutions of various typewriter-like output devices. TROFF rounds horizontal/vertical numerical parameter input to the actual horizontal/vertical resolution of the Graphic Systems typesetter. NROFF similarly rounds numerical input to the actual resolution of the output device indicated by the **−T** option (default Model 37 Teletype).

1.3. Numerical parameter input. Both NROFF and TROFF accept numerical input with the appended scale indicators shown in the following table, where *S* is the current type size in points, *V* is the current vertical line spacing in basic units, and *C* is a *nominal character width* in basic units.

Scale Indicator	Meaning	Number of basic units TROFF	NROFF
i	Inch	432	240
c	Centimeter	432×50/127	240×50/127
P	Pica = 1/6 inch	72	240/6
m	Em = *S* points	6×*S*	*C*
n	En = Em/2	3×*S*	*C, same as Em*
p	Point = 1/72 inch	6	240/72
u	Basic unit	1	1
v	Vertical line space	*V*	*V*
none	Default, see below		

In NROFF, *both* the em and the en are taken to be equal to the *C*, which is output-device dependent; common values are 1/10 and 1/12 inch. Actual character widths in NROFF need not be all the same and constructed characters such as −> (→) are often extra wide. The default scaling is ems for the horizontally-oriented requests and functions **ll**, **in**, **ti**, **ta**, **lt**, **po**, **mc**, **\h**, and **\l**; *V*s for the vertically-oriented requests and functions **pl**, **wh**, **ch**, **dt**, **sp**, **sv**, **ne**, **rt**, **\v**, **\x**, and **\L**; **p** for the **vs** request; and **u** for the requests **nr**, **if**, and **ie**. *All* other requests ignore any scale indicators. When a number register containing an already appropriately scaled number is interpolated to provide numerical input, the unit scale indicator **u** may need to be appended to prevent an additional inappropriate default scaling. The number, *N*, may be specified in decimal-fraction form but the parameter finally stored is rounded to an integer number of basic units.

The *absolute position* indicator | may be prepended to a number *N* to generate the distance to the vertical or horizontal place *N*. For vertically-oriented requests and functions, |*N* becomes the distance in basic units from the current vertical place on the page or in a *diversion* (§7.4) to the the vertical place *N*. For *all* other requests and functions, |*N* becomes the distance from the current horizontal place on the *input* line to the horizontal place *N*. For example,

.sp |3.2c

will space *in the required direction* to 3.2 centimeters from the top of the page.

1.4. Numerical expressions. Wherever numerical input is expected an expression involving parentheses, the arithmetic operators +, −, /, ∗, **%** (mod), and the logical operators <, >, <=, >=, = (or ==), **&** (and), **:** (or) may be used. Except where controlled by parentheses, evaluation of expressions is left-to-right; there is no operator precedence. In the case of certain requests, an initial + or − is stripped and interpreted as an increment or decrement indicator respectively. In the presence of default scaling, the desired scale indicator must be attached to *every* number in an expression for which the desired and default scaling differ. For example, if the number register **x** contains 2 and the current point size is 10, then

.ll (4.25i+\nxP+3)⁄2u

will set the line length to 1⁄2 the sum of 4.25 inches + 2 picas + 30 points.

1.5. Notation. Numerical parameters are indicated in this manual in two ways. ±*N* means that the argument may take the forms *N*, +*N*, or −*N* and that the corresponding effect is to set the affected parameter to *N*, to increment it by *N*, or to decrement it by *N* respectively. Plain *N* means that an initial algebraic sign is *not* an increment indicator, but merely the sign of *N*. Generally, unreasonable numerical input is either ignored or truncated to a reasonable value. For example, most requests expect to set parameters to non-negative values; exceptions are **sp**, **wh**, **ch**, **nr**, and **if**. The requests **ps**, **ft**, **po**, **vs**, **ls**, **ll**, **in**, and **lt** restore the *previous* parameter value in the *absence* of an argument.

Single character arguments are indicated by single lower case letters and one/two character arguments are indicated by a pair of lower case letters. Character string arguments are indicated by multi-character mnemonics.

2. Font and Character Size Control

2.1. Character set. The TROFF character set consists of the Graphics Systems Commercial II character set plus a Special Mathematical Font character set—each having 102 characters. These character sets are shown in the attached Table I. All ASCII characters are included, with some on the Special Font. With three exceptions, the ASCII characters are input as themselves, and non-ASCII characters are input in the form \(*xx* where *xx* is a two-character name given in the attached Table II. The three ASCII exceptions are mapped as follows:

ASCII Input		Printed by TROFF	
Character	Name	Character	Name
´	acute accent	’	close quote
`	grave accent	‘	open quote
−	minus	-	hyphen

The characters ´, `, and − may be input by \´, \`, and \− respectively or by their names (Table II). The ASCII characters @, #, ", ´, `, <, >, \, {, }, ˜, ˆ, and _ exist only on the Special Font and are printed as a 1-em space if that Font is not mounted.

NROFF understands the entire TROFF character set, but can in general print only ASCII characters, additional characters as may be available on the output device, such characters as may be able to be constructed by overstriking or other combination, and those that can reasonably be mapped into other printable characters. The exact behavior is determined by a driving table prepared for each device. The characters ´, `, and _ print as themselves.

2.2. Fonts. The default mounted fonts are Times Roman (**R**), Times Italic (**I**), Times Bold (**B**), and the Special Mathematical Font (**S**) on physical typesetter positions 1, 2, 3, and 4 respectively. These fonts are used in this document. The *current* font, initially Roman, may be changed (among the mounted fonts) by use of the **ft** request, or by imbedding at any desired point either **\f***x*, **\f(***xx*, or **\f***N* where *x* and *xx* are the name of a mounted font and *N* is a numerical font position. It is *not* necessary to change to the Special font; characters on that font are automatically handled. A request for a named but not-mounted font is *ignored*. TROFF can be informed that any particular font is mounted by use of the **fp** request. The list of known fonts is installation dependent. In the subsequent discussion of font-related requests, *F* represents either a one/two-character font name or the numerical font position, 1-4. The current font is available (as numerical position) in the read-only number register **.f**.

NROFF understands font control and normally underlines Italic characters (see §10.5).

2.3. Character size. Character point sizes available on the Graphic Systems typesetter are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. This is a range of 1/12 inch to 1/2 inch. The **ps** request is used to change or restore the point size. Alternatively the point size may be changed between any two characters by imbedding a **\s***N* at the desired point to set the size to *N*, or a **\s**±*N* (1≤*N*≤9) to increment/decrement the size by *N*; **\s0** restores the *previous* size. Requested point size values that are between two valid sizes yield the larger of the two. The current size is available in the **.s** register. NROFF ignores type size control.

Request Form	*Initial Value*	*If No Argument*	*Notes**	*Explanation*
.ps ±*N*	10 point	previous	E	Point size set to ±*N*. Alternatively imbed **\s***N* or **\s**±*N*. Any positive size value may be requested; if invalid, the next larger valid size will result, with a maximum of 36. A paired sequence +*N*, −*N* will work because the previous requested value is also remembered. Ignored in NROFF.
.ss *N*	12/36 em	ignored	E	Space-character size is set to *N*/36 ems. This size is the minimum word spacing in adjusted text. Ignored in NROFF.
.cs *F N M*	off	-	P	Constant character space (width) mode is set on for font *F* (if mounted); the width of every character will be taken to be *N*/36 ems. If *M* is absent, the em is that of the character's point size; if *M* is given, the em is *M*-points. All affected characters are centered in this space, including those with an actual width larger than this space. Special Font characters occurring while the current font is *F* are also so treated. If *N* is absent, the mode is turned off. The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.
.bd *F N*	off	-	P	The characters in font *F* will be artificially emboldened by printing each one twice, separated by *N*−1 basic units. A reasonable value for *N* is 3 when the character size is in the vicinity of 10 points. If *N* is missing the embolden mode is turned off. The column heads above were printed with **.bd I 3**. The mode must be still or again in effect when the characters are physically printed. Ignored in NROFF.
.bd S *F N*	off	-	P	The characters in the Special Font will be emboldened whenever the current font is *F*. This manual was printed with **.bd S B 3**. The mode must be still or again in effect when the characters are physically printed.
.ft *F*	Roman	previous	E	Font changed to *F*. Alternatively, imbed **\f***F*. The font name **P** is reserved to mean the previous font.
.fp *N F*	R,I,B,S	ignored	-	Font position. This is a statement that a font named *F* is mounted on position *N* (1-4). It is a fatal error if *F* is not known. The phototypesetter has four fonts physically mounted. Each font consists of a film strip which can be

*Notes are explained at the end of the Summary and Index above.

mounted on a numbered quadrant of a wheel. The default mounting sequence assumed by TROFF is R, I, B, and S on positions 1, 2, 3 and 4.

3. Page control

Top and bottom margins are *not* automatically provided; it is conventional to define two *macros* and to set *traps* for them at vertical positions 0 (top) and −*N* (*N* from the bottom). See §7 and Tutorial Examples §T2. A pseudo-page transition onto the *first* page occurs either when the first *break* occurs or when the first *non-diverted* text processing occurs. Arrangements for a trap to occur at the top of the first page must be completed before this transition. In the following, references to the *current diversion* (§7.4) mean that the mechanism being described works during both ordinary and diverted output (the former considered as the top diversion level).

The useable page width on the Graphic Systems phototypesetter is about 7.54 inches, beginning about 1⁄27 inch from the left edge of the 8 inch wide, continuous roll paper. The physical limitations on NROFF output are output-device dependent.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.pl ±*N*	11 in	11 in	**v**	Page length set to ±*N*. The internal limitation is about 75 inches in TROFF and about 136 inches in NROFF. The current page length is available in the **.p** register.
.bp ±*N*	*N*=1	-	B*,**v**	Begin page. The current page is ejected and a new page is begun. If ±*N* is given, the new page number will be ±*N*. Also see request **ns**.
.pn ±*N*	*N*=1	ignored	-	Page number. The next page (when it occurs) will have the page number ±*N*. A **pn** must occur before the initial pseudo-page transition to effect the page number of the first page. The current page number is in the **%** register.
.po ±*N*	0; 26⁄27 in†	previous	**v**	Page offset. The current *left margin* is set to ±*N*. The TROFF initial value provides about 1 inch of paper margin including the physical typesetter margin of 1⁄27 inch. In TROFF the maximum (line-length)+(page-offset) is about 7.54 inches. See §6. The current page offset is available in the **.o** register.
.ne *N*	-	*N*=1 *V*	D,**v**	Need *N* vertical space. If the distance, *D*, to the next trap position (see §7.5) is less than *N*, a forward vertical space of size *D* occurs, which will spring the trap. If there are no remaining traps on the page, *D* is the distance to the bottom of the page. If *D* < *V*, another line could still be output and spring the trap. In a diversion, *D* is the distance to the *diversion trap*, if any, or is very large.
.mk *R*	none	internal	D	Mark the *current* vertical place in an internal register (both associated with the current diversion level), or in register *R*, if given. See **rt** request.
.rt ±*N*	none	internal	D,**v**	Return *upward only* to a marked vertical place in the current diversion. If ±*N* (w.r.t. current place) is given, the place is ±*N* from the top of the page or diversion or, if *N* is absent, to a place marked by a previous **mk**. Note that the **sp** request (§5.3) may be used in all cases instead of **rt** by spacing to the absolute place stored in a explicit register; e. g. using the sequence **.mk** *R* ... **.sp** **|\n***R***u**.

*The use of " ´ " as control character (instead of "**.**") suppresses the break function.

†Values separated by ";" are for NROFF and TROFF respectively.

4. Text Filling, Adjusting, and Centering

4.1. Filling and adjusting. Normally, words are collected from input text lines and assembled into a output text line until some word doesn't fit. An attempt is then made the hyphenate the word in effort to assemble a part of it into the output line. The spaces between the words on the output line are then increased to spread out the line to the current *line length* minus any current *indent*. A *word* is any string of characters delimited by the *space* character or the beginning/end of the input line. Any adjacent pair of words that must be kept together (neither split across output lines nor spread apart in the adjustment process) can be tied together by separating them with the *unpaddable space* character "\ " (backslash-space). The adjusted word spacings are uniform in TROFF and the minimum interword spacing can be controlled with the **ss** request (§2). In NROFF, they are normally nonuniform because of quantization to character-size spaces; however, the command line option **−e** causes uniform spacing with full output device resolution. Filling, adjustment, and hyphenation (§13) can all be prevented or controlled. The *text length* on the last line output is available in the **.n** register, and text base-line position on the page for this line is in the **nl** register. The text base-line high-water mark (lowest place) on the current page is in the **.h** register.

An input text line ending with **.**, **?**, or **!** is taken to be the end of a *sentence*, and an additional space character is automatically provided during filling. Multiple inter-word space characters found in the input are retained, except for trailing spaces; initial spaces also cause a *break*.

When filling is in effect, a **\p** may be imbedded or attached to a word to cause a *break* at the *end* of the word and have the resulting output line *spread out* to fill the current line length.

A text input line that happens to begin with a control character can be made to not look like a control line by prefacing it with the non-printing, zero-width filler character **\&**. Still another way is to specify output translation of some convenient character into the control character using **tr** (§10.5).

4.2. Interrupted text. The copying of a input line in *nofill* (non-fill) mode can be *interrupted* by terminating the partial line with a **\c**. The *next* encountered input text line will be considered to be a continuation of the same line of input text. Similarly, a word within *filled* text may be interrupted by terminating the word (and line) with **\c**; the next encountered text will be taken as a continuation of the interrupted word. If the intervening control lines cause a break, any partial line will be forced out along with any partial word.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.br	-	-	B	Break. The filling of the line currently being collected is stopped and the line is output without adjustment. Text lines beginning with space characters and empty text lines (blank lines) also cause a break.
.fi	fill on	-	B,E	Fill subsequent output lines. The register **.u** is 1 in fill mode and 0 in nofill mode.
.nf	fill on	-	B,E	Nofill. Subsequent output lines are *neither* filled *nor* adjusted. Input text lines are copied directly to output lines *without regard* for the current line length.
.ad *c*	adj,both	adjust	E	Line adjustment is begun. If fill mode is not on, adjustment will be deferred until fill mode is back on. If the type indicator *c* is present, the adjustment type is changed as shown in the following table.

Indicator	Adjust Type
l	adjust left margin only
r	adjust right margin only
c	center
b or **n**	adjust both margins
absent	unchanged

.na	adjust	-	E	Noadjust. Adjustment is turned off; the right margin will be ragged. The adjustment type for **ad** is not changed. Output line filling still occurs if fill mode is on.
.ce *N*	off	*N*=1	B,E	Center the next *N* input text lines within the current (line-length minus indent). If *N*=0, any residual count is cleared. A break occurs after each of the *N* input lines. If the input line is too long, it will be left adjusted.

5. Vertical Spacing

5.1. Base-line spacing. The vertical spacing *(V)* between the base-lines of successive output lines can be set using the **vs** request with a resolution of 1⁄144 inch = 1⁄2 point in TROFF, and to the output device resolution in NROFF. *V* must be large enough to accommodate the character sizes on the affected output lines. For the common type sizes (9-12 points), usual typesetting practice is to set *V* to 2 points greater than the point size; TROFF default is 10-point type on a 12-point spacing (as in this document). The current *V* is available in the **.v** register. Multiple-*V* line separation (e. g. double spacing) may be requested with **ls**.

5.2. Extra line-space. If a word contains a vertically tall construct requiring the output line containing it to have extra vertical space before and⁄or after it, the *extra-line-space* function **\x**´*N* ´ can be imbedded in or attached to that word. In this and other functions having a pair of delimiters around their parameter (here ´), the delimiter choice is arbitrary, except that it can't look like the continuation of a number expression for *N*. If *N* is negative, the output line containing the word will be preceded by *N* extra vertical space; if *N* is positive, the output line containing the word will be followed by *N* extra vertical space. If successive requests for extra space apply to the same line, the maximum values are used. The most recently utilized post-line extra line-space is available in the **.a** register.

5.3. Blocks of vertical space. A block of vertical space is ordinarily requested using **sp**, which honors the *no-space* mode and which does not space *past* a trap. A contiguous block of vertical space may be reserved using **sv**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.vs *N*	1⁄6in;12pts	previous	E,**p**	Set vertical base-line spacing size *V*. Transient *extra* vertical space available with **\x**´*N* ´ (see above).
.ls *N*	*N*=1	previous	E	*Line* spacing set to ±*N*. *N*−1 *V*s *(blank lines)* are appended to each output text line. Appended blank lines are omitted, if the text or previous appended blank line reached a trap position.
.sp *N*	-	*N*=1*V*	B,**v**	Space vertically in *either* direction. If *N* is negative, the motion is *backward* (upward) and is limited to the distance to the top of the page. Forward (downward) motion is truncated to the distance to the nearest trap. If the no-space mode is on, no spacing occurs (see **ns**, and **rs** below).
.sv *N*	-	*N*=1*V*	**v**	Save a contiguous vertical block of size *N*. If the distance to the next trap is greater than *N*, *N* vertical space is output. No-space mode has *no* effect. If this distance is less than *N*, no vertical space is immediately output, but *N* is remembered for later output (see **os**). Subsequent **sv** requests will overwrite any still remembered *N*.
.os	-	-	-	Output saved vertical space. No-space mode has *no* effect. Used to finally output a block of vertical space requested by an earlier **sv** request.
.ns	space	-	D	No-space mode turned on. When on, the no-space mode inhibits **sp** requests and **bp** requests *without* a next page number. The no-space mode is turned off when a line of output occurs, or with **rs**.

.rs	space	-	D	Restore spacing. The no-space mode is turned off.
Blank text line.		-	B	Causes a break and output of a blank line exactly like **sp 1**.

6. Line Length and Indenting

The maximum line length for fill mode may be set with **ll**. The indent may be set with **in**; an indent applicable to *only* the *next* output line may be set with **ti**. The line length includes indent space but *not* page offset space. The line-length minus the indent is the basis for centering with **ce**. The effect of **ll**, **in**, or **ti** is delayed, if a partially collected line exists, until after that line is output. In fill mode the length of text on an output line is less than or equal to the line length minus the indent. The current line length and indent are available in registers **.l** and **.i** respectively. The length of *three-part titles* produced by **tl** (see §14) is *independently* set by **lt**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ll ±*N*	6.5 in	previous	E,**m**	Line length is set to ±*N*. In TROFF the maximum (line-length)+(page-offset) is about 7.54 inches.
.in ±*N*	*N*=0	previous	B,E,**m**	Indent is set to ±*N*. The indent is prepended to each output line.
.ti ±*N*	-	ignored	B,E,**m**	Temporary indent. The *next* output text line will be indented a distance ±*N* with respect to the current indent. The resulting total indent may not be negative. The current indent is not changed.

7. Macros, Strings, Diversion, and Position Traps

7.1. Macros and strings. A *macro* is a named set of arbitrary *lines* that may be invoked by name or with a *trap*. A *string* is a named string of *characters*, *not* including a newline character, that may be interpolated by name at any point. Request, macro, and string names share the *same* name list. Macro and string names may be one or two characters long and may usurp previously defined request, macro, or string names. Any of these entities may be renamed with **rn** or removed with **rm**. Macros are created by **de** and **di**, and appended to by **am** and **da**; **di** and **da** cause normal output to be stored in a macro. Strings are created by **ds** and appended to by **as**. A macro is invoked in the same way as a request; a control line beginning **.***xx* will interpolate the contents of macro *xx*. The remainder of the line may contain up to nine *arguments*. The strings *x* and *xx* are interpolated at any desired point with **x* and *(*xx* respectively. String references and macro invocations may be nested.

7.2. Copy mode input interpretation. During the definition and extension of strings and macros (not by diversion) the input is read in *copy mode*. The input is copied without interpretation *except* that:

- The contents of number registers indicated by **\n** are interpolated.
- Strings indicated by * are interpolated.
- Arguments indicated by **\$** are interpolated.
- Concealed newlines indicated by \(newline) are eliminated.
- Comments indicated by \" are eliminated.
- **\t** and **\a** are interpreted as ASCII horizontal tab and SOH respectively (§9).
- \\ is interpreted as \.
- **\.** is interpreted as "**.**".

These interpretations can be suppressed by prepending a \. For example, since \\ maps into a \, **n** will copy as **\n** which will be interpreted as a number register indicator when the macro or string is reread.

7.3. Arguments. When a macro is invoked by name, the remainder of the line is taken to contain up to nine arguments. The argument separator is the space character, and arguments may be surrounded by double-quotes to permit imbedded space characters. Pairs of double-quotes may be imbedded in double-quoted arguments to represent a single double-quote. If the desired arguments won't fit on a line, a concealed newline may be used to continue on the next line.

When a macro is invoked the *input level* is *pushed down* and any arguments available at the previous level become unavailable until the macro is completely read and the previous level is restored. A macro's own arguments can be interpolated at *any* point within the macro with **\$***N*, which interpolates the *N*th argument (1≤*N*≤9).

If an invoked argument doesn't exist, a null string results. For example, the macro *xx* may be defined by

```
.de xx      \"begin definition
Today is \\$1 the \\$2.
..          \"end definition
```

and called by

```
.xx Monday 14th
```

to produce the text

Today is Monday the 14th.

Note that the **\$** was concealed in the definition with a prepended \. The number of currently available arguments is in the **.$** register.

No arguments are available at the top (non-macro) level in this implementation. Because string referencing is implemented as a input-level push down, no arguments are available from *within* a string. No arguments are available within a trap-invoked macro.

Arguments are copied in *copy mode* onto a stack where they are available for reference. The mechanism does not allow an argument to contain a direct reference to a *long* string (interpolated at copy time) and it is advisable to conceal string references (with an extra \ ) to delay interpolation until argument reference time.

7.4. Diversions. Processed output may be diverted into a macro for purposes such as footnote processing (see Tutorial §T5) or determining the horizontal and vertical size of some text for conditional changing of pages or columns. A single diversion trap may be set at a specified vertical position. The number registers **dn** and **dl** respectively contain the vertical and horizontal size of the most recently ended diversion. Processed text that is diverted into a macro retains the vertical size of each of its lines when reread in *nofill* mode regardless of the current *V*. Constant-spaced (**cs**) or emboldened (**bd**) text that is diverted can be reread correctly only if these modes are again or still in effect at reread time. One way to do this is to imbed in the diversion the appropriate **cs** or **bd** requests with the *transparent* mechanism described in §10.6.

Diversions may be nested and certain parameters and registers are associated with the current diversion level (the top non-diversion level may be thought of as the 0th diversion level). These are the diversion trap and associated macro, no-space mode, the internally-saved marked place (see **mk** and **rt**), the current vertical place (**.d** register), the current high-water text base-line (**.h** register), and the current diversion name (**.z** register).

7.5. Traps. Three types of trap mechanisms are available—page traps, a diversion trap, and an input-line-count trap. Macro-invocation traps may be planted using **wh** at any page position including the top. This trap position may be changed using **ch**. Trap positions at or below the bottom of the page have no effect unless or until moved to within the page or rendered effective by an increase in page length. Two traps may be planted at the *same* position only by first planting them at different positions and then moving one of the traps; the first planted trap will conceal the second unless and until the first one is moved (see Tutorial Examples §T5). If the first one is moved back, it again conceals the second trap. The macro associated with a page trap is automatically invoked when a line of text is output whose vertical size *reaches* or *sweeps past* the trap position. Reaching the bottom of a page springs the top-of-page trap, if any, provided there is a next page. The distance to the next trap position is available in the **.t** register; if there are no traps between the current position and the bottom of the page, the distance returned is the distance to the page bottom.

A macro-invocation trap effective in the current diversion may be planted using **dt**. The **.t** register works in a diversion; if there is no subsequent trap a *large* distance is returned. For a description of input-line-count traps, see **it** below.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.de *xx yy*	-	*.yy*=**..**	-	Define or redefine the macro *xx*. The contents of the macro begin on the next input line. Input lines are copied in *copy mode* until the definition is terminated by a line beginning with **.***yy*, whereupon the macro *yy* is called. In the absence of *yy*, the definition is terminated by a line beginning with "**..**".

				A macro may contain **de** requests provided the terminating macros differ or the contained definition terminator is concealed. "**..**" can be concealed as **..** which will copy as **..** and be reread as "**..**".
.am *xx yy*	-	*.yy*=**..**	-	Append to macro (append version of **de**).
.ds *xx string*	-	ignored	-	Define a string *xx* containing *string*. Any initial double-quote in *string* is stripped off to permit initial blanks.
.as *xx string*	-	ignored	-	Append *string* to string *xx* (append version of **ds**).
.rm *xx*	-	ignored	-	Remove request, macro, or string. The name *xx* is removed from the name list and any related storage space is freed. Subsequent references will have no effect.
.rn *xx yy*	-	ignored	-	Rename request, macro, or string *xx* to *yy*. If *yy* exists, it is first removed.
.di *xx*	-	end	D	Divert output to macro *xx*. Normal text processing occurs during diversion except that page offsetting is not done. The diversion ends when the request **di** or **da** is encountered without an argument; extraneous requests of this type should not appear when nested diversions are being used.
.da *xx*	-	end	D	Divert, appending to *xx* (append version of **di**).
.wh *N xx*	-	-	**v**	Install a trap to invoke *xx* at page position *N;* a *negative N* will be interpreted with respect to the page *bottom*. Any macro previously planted at *N* is replaced by *xx*. A zero *N* refers to the *top* of a page. In the absence of *xx*, the first found trap at *N*, if any, is removed.
.ch *xx N*	-	-	**v**	Change the trap position for macro *xx* to be *N*. In the absence of *N*, the trap, if any, is removed.
.dt *N xx*	-	off	D,**v**	Install a diversion trap at position *N* in the *current* diversion to invoke macro *xx*. Another **dt** will redefine the diversion trap. If no arguments are given, the diversion trap is removed.
.it *N xx*	-	off	E	Set an input-line-count trap to invoke the macro *xx* after *N* lines of *text* input have been read (control or request lines don't count). The text may be in-line text or text interpolated by inline or trap-invoked macros.
.em *xx*	none	none	-	The macro *xx* will be invoked when all input has ended. The effect is the same as if the contents of *xx* had been at the end of the last file processed.

8. Number Registers

A variety of parameters are available to the user as predefined, named *number registers* (see Summary and Index, page 7). In addition, the user may define his own named registers. Register names are one or two characters long and *do not* conflict with request, macro, or string names. Except for certain predefined read-only registers, a number register can be read, written, automatically incremented or decremented, and interpolated into the input in a variety of formats. One common use of user-defined registers is to automatically number sections, paragraphs, lines, etc. A number register may be used any time numerical input is expected or desired and may be used in numerical *expressions* (§1.4).

Number registers are created and modified using **nr**, which specifies the name, numerical value, and the auto-increment size. Registers are also modified, if accessed with an auto-incrementing sequence. If the registers *x* and *xx* both contain *N* and have the auto-increment size *M*, the following access sequences have the effect shown:

Sequence	Effect on Register	Value Interpolated
\n$x$	none	N
\n(xx	none	N
\n+x	x incremented by M	$N{+}M$
\n–x	x decremented by M	$N{-}M$
\n+(xx	xx incremented by M	$N{+}M$
\n–(xx	xx decremented by M	$N{-}M$

When interpolated, a number register is converted to decimal (default), decimal with leading zeros, lower-case Roman, upper-case Roman, lower-case sequential alphabetic, or upper-case sequential alphabetic according to the format specified by **af**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nr R $\pm N$ M	-	**u**		The number register R is assigned the value $\pm N$ with respect to the previous value, if any. The increment for auto-incrementing is set to M.
.af R c	arabic	-	-	Assign format c to register R. The available formats are:

Format	Numbering Sequence
1	0,1,2,3,4,5,...
001	000,001,002,003,004,005,...
i	0,i,ii,iii,iv,v,...
I	0,I,II,III,IV,V,...
a	0,a,b,c,...,z,aa,ab,...,zz,aaa,...
A	0,A,B,C,...,Z,AA,AB,...,ZZ,AAA,...

An arabic format having N digits specifies a field width of N digits (example 2 above). The read-only registers and the *width* function (§11.2) are always arabic.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.rr R	-	ignored	-	Remove register R. If many registers are being created dynamically, it may become necessary to remove no longer used registers to recapture internal storage space for newer registers.

9. Tabs, Leaders, and Fields

9.1. Tabs and leaders. The ASCII horizontal tab character and the ASCII SOH (hereafter known as the *leader* character) can both be used to generate either horizontal motion or a string of repeated characters. The length of the generated entity is governed by internal *tab stops* specifiable with **ta**. The default difference is that tabs generate motion and leaders generate a string of periods; **tc** and **lc** offer the choice of repeated character or motion. There are three types of internal tab stops—*left* adjusting, *right* adjusting, and *centering*. In the following table: D is the distance from the current position on the *input* line (where a tab or leader was found) to the next tab stop; *next-string* consists of the input characters following the tab (or leader) up to the next tab (or leader) or end of line; and W is the width of *next-string*.

Tab type	Length of motion or repeated characters	Location of *next-string*
Left	D	Following D
Right	$D-W$	Right adjusted within D
Centered	$D-W/2$	Centered on right end of D

The length of generated motion is allowed to be negative, but that of a repeated character string cannot be. Repeated character strings contain an integer number of characters, and any residual distance is prepended as motion. Tabs or leaders found after the last tab stop are ignored, but may be used as *next-string* terminators.

Tabs and leaders are not interpreted in *copy mode*. **\t** and **\a** always generate a non-interpreted tab and leader respectively, and are equivalent to actual tabs and leaders in *copy mode*.

9.2. Fields. A *field* is contained between a *pair* of *field delimiter* characters, and consists of sub-strings separated by *padding* indicator characters. The field length is the distance on the *input* line from the position where the field begins to the next tab stop. The difference between the total length of all the sub-strings and the field length is incorporated as horizontal padding space that is divided among the indicated padding places. The incorporated padding is allowed to be negative. For example, if the field delimiter is # and the padding indicator is ^, #*^xxx^right*# specifies a right-adjusted string with the string *xxx* centered in the remaining space.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ta *Nt* ...	0.8; 0.5in	none	E,**m**	Set tab stops and types. *t*=**R**, right adjusting; *t*=**C**, centering; *t* absent, left adjusting. TROFF tab stops are preset every 0.5in.; NROFF every 0.8in. The stop values are separated by spaces, and a value preceded by **+** is treated as an increment to the previous stop value.
.tc *c*	none	none	E	The tab repetition character becomes *c*, or is removed specifying motion.
.lc *c*	**.**	none	E	The leader repetition character becomes *c*, or is removed specifying motion.
.fc *a b*	off	off	-	The field delimiter is set to *a*; the padding indicator is set to the *space* character or to *b*, if given. In the absence of arguments the field mechanism is turned off.

10. Input and Output Conventions and Character Translations

10.1. Input character translations. Ways of inputting the graphic character set were discussed in §2.1. The ASCII control characters horizontal tab (§9.1), SOH (§9.1), and backspace (§10.3) are discussed elsewhere. The newline delimits input lines. In addition, STX, ETX, ENQ, ACK, and BEL are accepted, and may be used as delimiters or translated into a graphic with **tr** (§10.5). *All* others are ignored.

The *escape* character \ introduces *escape sequences*—causes the following character to mean another character, or to indicate some function. A complete list of such sequences is given in the Summary and Index on page 6. \ should not be confused with the ASCII control character ESC of the same name. The escape character \ can be input with the sequence \\. The escape character can be changed with **ec**, and all that has been said about the default \ becomes true for the new escape character. **\e** can be used to print whatever the current escape character is. If necessary or convenient, the escape mechanism may be turned off with **eo**, and restored with **ec**.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ec *c*	\	\	-	Set escape character to \, or to *c*, if given.
.eo	on	-	-	Turn escape mechanism off.

10.2. Ligatures. Five ligatures are available in the current TROFF character set — **fi**, **fl**, **ff**, **ffi**, and **ffl**. They may be input (even in NROFF) by **\(fi**, **\(fl**, **\(ff**, **\(Fi**, and **\(Fl** respectively. The ligature mode is normally on in TROFF, and *automatically* invokes ligatures during input.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.lg *N*	off; on	on	-	Ligature mode is turned on if *N* is absent or non-zero, and turned off if *N*=0. If *N*=2, only the two-character ligatures are automatically invoked. Ligature mode is inhibited for request,

				macro, string, register, or file names, and in *copy mode*. No effect in NROFF.

10.3. Backspacing, underlining, overstriking, etc. Unless in *copy mode*, the ASCII backspace character is replaced by a backward horizontal motion having the width of the space character. Underlining as a form of line-drawing is discussed in §12.4. A generalized overstriking function is described in §12.1.

NROFF automatically underlines characters in the *underline* font, specifiable with **uf**, normally that on font position 2 (normally Times Italic, see §2.2). In addition to **ft** and **\f***F*, the underline font may be selected by **ul** and **cu**. Underlining is restricted to an output-device-dependent subset of *reasonable* characters.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ul *N*	off	*N*=1	E	Underline in NROFF (italicize in TROFF) the next *N* input text lines. Actually, switch to *underline* font, saving the current font for later restoration; *other* font changes within the span of a **ul** will take effect, but the restoration will undo the last change. Output generated by **tl** (§14) *is* affected by the font change, but does *not* decrement *N*. If $N>1$, there is the risk that a trap interpolated macro may provide text lines within the span; environment switching can prevent this.
.cu *N*	off	*N*=1	E	A variant of **ul** that causes *every* character to be underlined in NROFF. Identical to **ul** in TROFF.
.uf *F*	Italic	Italic	-	Underline font set to *F*. In NROFF, *F* may *not* be on position 1 (initially Times Roman).

10.4. Control characters. Both the control character **.** and the *no-break* control character ´ may be changed, if desired. Such a change must be compatible with the design of any macros used in the span of the change, and particularly of any trap-invoked macros.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.cc *c*	**.**	**.**	E	The basic control character is set to *c*, or reset to "**.**".
.c2 *c*	´	´	E	The *nobreak* control character is set to *c*, or reset to "´".

10.5. Output translation. One character can be made a stand-in for another character using **tr**. All text processing (e. g. character comparisons) takes place with the input (stand-in) character which appears to have the width of the final character. The graphic translation occurs at the moment of output (including diversion).

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.tr *abcd....*	none	-	O	Translate *a* into *b*, *c* into *d*, etc. If an odd number of characters is given, the last one will be mapped into the space character. To be consistent, a particular translation must stay in effect from *input* to *output* time.

10.6. Transparent throughput. An input line beginning with a **\!** is read in *copy mode* and *transparently* output (without the initial **\!**); the text processor is otherwise unaware of the line's presence. This mechanism may be used to pass control information to a post-processor or to imbed control lines in a macro created by a diversion.

10.7. Comments and concealed newlines. An uncomfortably long input line that must stay one line (e. g. a string definition, or nofilled text) can be split into many physical lines by ending all but the last one with the escape \. The sequence \(newline) is *always* ignored—except in a comment. Comments may be imbedded at the *end* of any line by prefacing them with \". The newline at the end of a comment cannot be concealed. A line beginning with \" will appear as a blank line and behave like **.sp 1**; a comment can be on a line by itself by beginning the line with **.**\".

11. Local Horizontal and Vertical Motions, and the Width Function

11.1. Local Motions. The functions **\v**′*N*′ and **\h**′*N*′ can be used for *local* vertical and horizontal motion respectively. The distance *N* may be negative; the *positive* directions are *rightward* and *downward*. A *local* motion is one contained *within* a line. To avoid unexpected vertical dislocations, it is necessary that the *net* vertical local motion within a word in filled text and otherwise within a line balance to zero. The above and certain other escape sequences providing local motion are summarized in the following table.

Vertical Local Motion	Effect in TROFF	NROFF	Horizontal Local Motion	Effect in TROFF	NROFF
\v′*N*′	Move distance *N*		**\h**′*N*′	Move distance *N*	
			\(space)	Unpaddable space-size space	
\u	½ em up	½ line up	**\0**	Digit-size space	
\d	½ em down	½ line down			
\r	1 em up	1 line up	|	1⁄6 em space	ignored
			\^	1⁄12 em space	ignored

As an example, **E**^{2} could be generated by the sequence **E\s−2\v′−0.4m′2\v′0.4m′\s+2**; it should be noted in this example that the 0.4 em vertical motions are at the smaller size.

11.2. Width Function. The *width* function **\w**′*string*′ generates the numerical width of *string* (in basic units). Size and font changes may be safely imbedded in *string*, and will not affect the current environment. For example, **.ti −\w′1. ′u** could be used to temporarily indent leftward a distance equal to the size of the string "**1.** ".

The width function also sets three number registers. The registers **st** and **sb** are set respectively to the highest and lowest extent of *string* relative to the baseline; then, for example, the total *height* of the string is **\n(stu−\n(sbu**. In TROFF the number register **ct** is set to a value between 0 and 3: 0 means that all of the characters in *string* were short lower case characters without descenders (like **e**); 1 means that at least one character has a descender (like **y**); 2 means that at least one character is tall (like **H**); and 3 means that both tall characters and characters with descenders are present.

11.3. Mark horizontal place. The escape sequence **\k***x* will cause the *current* horizontal position in the *input line* to be stored in register *x*. As an example, the construction **\kx***word***\h′|\nxu+2u′***word* will embolden *word* by backing up to almost its beginning and overprinting it, resulting in ***word***.

12. Overstrike, Bracket, Line-drawing, and Zero-width Functions

12.1. Overstriking. Automatically centered overstriking of up to nine characters is provided by the *overstrike* function **\o**′*string*′. The characters in *string* overprinted with centers aligned; the total width is that of the widest character. *string* should *not* contain local vertical motion. As examples, **\o′e\′′** produces **é**, and **\o′\(mo\(sl′** produces ∉.

12.2. Zero-width characters. The function **\z***c* will output *c* without spacing over it, and can be used to produce left-aligned overstruck combinations. As examples, **\z\(ci\(pl** will produce ⊕ and **\(br\z\(rn\(ul\(br** will produce the smallest possible constructed box □ .

12.3. Large Brackets. The Special Mathematical Font contains a number of bracket construction pieces (⎧ ⎩ ⎫ ⎭ ⎨ ⎬ | ⌊ ⌋⌈ ⌉) that can be combined into various bracket styles. The function **\b**′*string*′ may be used to pile up vertically the characters in *string* (the first character on top and the last at the bottom); the characters are vertically separated by 1 em and the total pile is centered 1⁄2 em above the current baseline (½ line in NROFF). For example, **\b′\(lc\(lf′E\|\b′\(rc\(rf′\x′−0.5m′\x′0.5m′** produces [**E**].

12.4. Line drawing. The function **\l**′*Nc*′ will draw a string of repeated *c*'s towards the right for a distance *N*. (**\l** is \(lower case L). If *c* looks like a continuation of an expression for *N*, it may insulated from *N* with a **\&**. If *c* is not specified, the _ (baseline rule) is used (underline character in NROFF). If *N* is negative, a backward horizontal motion of size *N* is made *before* drawing the string. Any space resulting from *N*/(size of *c*) having a remainder is put at the beginning (left end) of the string. In the case of characters that are designed to be connected such as baseline-rule _, underrule _, and root-en ‾, the remainder space is covered by over-lapping. If *N* is *less* than the width of *c*, a single *c* is centered on a distance *N*. As an example, a macro to underscore a string can be written

```
.de us
\\$1\l´|0\(ul´
..
```

or one to draw a box around a string

```
.de bx
\(br\|\\$1\|\(br\l´|0\(rn´\l´|0\(ul´
..
```

such that

.ul "underlined words"

and

.bx "words in a box"

yield underlined words and words in a box.

The function **\L**´*Nc*´ will draw a vertical line consisting of the (optional) character *c* stacked vertically apart 1 em (1 line in NROFF), with the first two characters overlapped, if necessary, to form a continuous line. The default character is the *box rule* | (\\(**br**); the other suitable character is the *bold vertical* | (\\(**bv**). The line is begun without any initial motion relative to the current base line. A positive *N* specifies a line drawn downward and a negative *N* specifies a line drawn upward. After the line is drawn *no* compensating motions are made; the instantaneous baseline is at the *end* of the line.

The horizontal and vertical line drawing functions may be used in combination to produce large boxes. The zero-width *box-rule* and the ½-em wide *underrule* were *designed* to form corners when using 1-em vertical spacings. For example the macro

```
.de eb
.sp –1          \"compensate for next automatic base-line spacing
.nf             \"avoid possibly overflowing word buffer
\h´–.5n´\L´|\\nau–1´\l´\\n(.lu+1n\(ul´\L´–|\\nau+1´\l´|0u–.5n\(ul´    \"draw box
.fi
..
```

will draw a box around some text whose beginning vertical place was saved in number register *a* (e. g. using **.mk a**) as done for this paragraph.

13. Hyphenation.

The automatic hyphenation may be switched off and on. When switched on with **hy**, several variants may be set. A *hyphenation indicator* character may be imbedded in a word to specify desired hyphenation points, or may be prepended to suppress hyphenation. In addition, the user may specify a small exception word list.

Only words that consist of a central alphabetic string surrounded by (usually null) non-alphabetic strings are considered candidates for automatic hyphenation. Words that were input containing hyphens (minus), em-dashes (\\(**em**), or hyphenation indicator characters—such as mother-in-law—are *always* subject to splitting after those characters, whether or not automatic hyphenation is on or off.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nh	hyphenate	-	E	Automatic hyphenation is turned off.
.hy*N*	on,*N*=1	on,*N*=1	E	Automatic hyphenation is turned on for $N \geq 1$, or off for $N = 0$. If $N = 2$, *last* lines (ones that will cause a trap) are not hyphenated. For $N = 4$ and 8, the last and first two characters respectively of a word are not split off. These values are additive; i. e. $N = 14$ will invoke all three restrictions.

.hc *c*	\%	\%	E	Hyphenation indicator character is set to *c* or to the default **\%**. The indicator does not appear in the output.
.hw *word1* ...	ignored	-		Specify hyphenation points in words with imbedded minus signs. Versions of a word with terminal *s* are implied; i. e. *dig−it* implies *dig−its*. This list is examined initially *and* after each suffix stripping. The space available is small—about 128 characters.

14. Three Part Titles.

The titling function **tl** provides for automatic placement of three fields at the left, center, and right of a line with a title-length specifiable with **lt**. **tl** may be used anywhere, and is independent of the normal text collecting process. A common use is in header and footer macros.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.tl *´left´center´right´*		-	-	The strings *left*, *center*, and *right* are respectively left-adjusted, centered, and right-adjusted in the current title-length. Any of the strings may be empty, and overlapping is permitted. If the page-number character (initially **%**) is found within any of the fields it is replaced by the current page number having the format assigned to register **%**. Any character may be used as the string delimiter.
.pc *c*	%	off	-	The page number character is set to *c*, or removed. The page-number register remains **%**.
.lt ±*N*	6.5 in	previous	E,**m**	Length of title set to ±*N*. The line-length and the title-length are *independent*. Indents do not apply to titles; page-offsets do.

15. Output Line Numbering.

Automatic sequence numbering of output lines may be requested with **nm**. When in effect, a three-digit,
arabic number plus a digit-space is prepended to output text lines. The text lines are thus offset by four
3 digit-spaces, and otherwise retain their line length; a reduction in line length may be desired to keep the
right margin aligned with an earlier margin. Blank lines, other vertical spaces, and lines generated by **tl** are
not numbered. Numbering can be temporarily suspended with **nn**, or with an **.nm** followed by a later
6 **.nm +0**. In addition, a line number indent *I*, and the number-text separation *S* may be specified in digit-
spaces. Further, it can be specified that only those line numbers that are multiples of some number *M* are to
be printed (the others will appear as blank number fields).

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.nm ±*N M S I*		off	E	Line number mode. If ±*N* is given, line numbering is turned on, and the next output line numbered is numbered ±*N*. Default values are *M*=1, *S*=1, and *I*=0. Parameters corresponding to missing arguments are unaffected; a non-numeric argument is considered missing. In the absence of all arguments, numbering is turned off; the next line number is preserved for possible further use in number register **ln**.
.nn *N*	-	*N*=1	E	The next *N* text output lines are not numbered.

9 As an example, the paragraph portions of this section are numbered with *M*=3: **.nm 1 3** was placed at the
beginning; **.nm** was placed at the end of the first paragraph; and **.nm +0** was placed in front of this para-
graph; and **.nm** finally placed at the end. Line lengths were also changed (by **\w´0000´u**) to keep the right
12 side aligned. Another example is **.nm +5 5 x 3** which turns on numbering with the line number of the next
line to be 5 greater than the last numbered line, with *M*=5, with spacing *S* untouched, and with the indent *I*

set to 3.

16. Conditional Acceptance of Input

In the following, *c* is a one-character, built-in *condition* name, **!** signifies *not*, *N* is a numerical expression, *string1* and *string2* are strings delimited by any non-blank, non-numeric character *not* in the strings, and *anything* represents what is conditionally accepted.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.if *c anything*		-	-	If condition *c* true, accept *anything* as input; in multi-line case use \\{*anything*\\}.
.if !*c anything*		-	-	If condition *c* false, accept *anything*.
.if *N anything*	-		u	If expression *N* > 0, accept *anything*.
.if !*N anything*		-	u	If expression *N* ≤ 0, accept *anything*.
.if *´string1´string2´ anything*			-	If *string1* identical to *string2*, accept *anything*.
.if !*´string1´string2´ anything*			-	If *string1* not identical to *string2*, accept *anything*.
.ie *c anything*	-		u	If portion of if-else; all above forms (like **if**).
.el *anything*		-	-	Else portion of if-else.

The built-in condition names are:

Condition Name	True If
o	Current page number is odd
e	Current page number is even
t	Formatter is TROFF
n	Formatter is NROFF

If the condition *c* is *true*, or if the number *N* is greater than zero, or if the strings compare identically (including motions and character size and font), *anything* is accepted as input. If a **!** precedes the condition, number, or string comparison, the sense of the acceptance is reversed.

Any spaces between the condition and the beginning of *anything* are skipped over. The *anything* can be either a single input line (text, macro, or whatever) or a number of input lines. In the multi-line case, the first line must begin with a left delimiter \{ and the last line must end with a right delimiter \}.

The request **ie** (if-else) is identical to **if** except that the acceptance state is remembered. A subsequent and matching **el** (else) request then uses the reverse sense of that state. **ie - el** pairs may be nested.

Some examples are:

```
.if e .tl ´Even Page %´´´
```

which outputs a title if the page number is even; and

```
.ie \n%>1 \{\
´sp 0.5i
.tl ´Page %´´´
´sp |1.2i \}
.el .sp |2.5i
```

which treats page 1 differently from other pages.

17. Environment Switching.

A number of the parameters that control the text processing are gathered together into an *environment*, which can be switched by the user. The environment parameters are those associated with requests noting E in their *Notes* column; in addition, partially collected lines and words are in the environment. Everything else is global;

examples are page-oriented parameters, diversion-oriented parameters, number registers, and macro and string definitions. All environments are initialized with default parameter values.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.ev *N*	*N*=0	previous	-	Environment switched to environment 0≤*N*≤2. Switching is done in push-down fashion so that restoring a previous environment *must* be done with **.ev** rather than specific reference.

18. Insertions from the Standard Input

The input can be temporarily switched to the system *standard input* with **rd**, which will switch back when *two* newlines in a row are found (the *extra* blank line is not used). This mechanism is intended for insertions in form-letter-like documentation. On UNIX, the *standard input* can be the user's keyboard, a *pipe*, or a *file*.

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.rd *prompt*	-	*prompt*=BEL	-	Read insertion from the standard input until two newlines in a row are found. If the standard input is the user's keyboard, *prompt* (or a BEL) is written onto the user's terminal. **rd** behaves like a macro, and arguments may be placed after *prompt*.
.ex	-	-	-	Exit from NROFF/TROFF. Text processing is terminated exactly as if all input had ended.

If insertions are to be taken from the terminal keyboard *while* output is being printed on the terminal, the command line option −**q** will turn off the echoing of keyboard input and prompt only with BEL. The regular input and insertion input *cannot* simultaneously come from the standard input.

As an example, multiple copies of a form letter may be prepared by entering the insertions for all the copies in one file to be used as the standard input, and causing the file containing the letter to reinvoke itself using **nx** (§19); the process would ultimately be ended by an **ex** in the insertion file.

19. Input/Output File Switching

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.so *filename*		-	-	Switch source file. The top input (file reading) level is switched to *filename*. The effect of an **so** encountered in a macro is not felt until the input level returns to the file level. When the new file ends, input is again taken from the original file. **so**'s may be nested.
.nx *filename*		end-of-file	-	Next file is *filename*. The current file is considered ended, and the input is immediately switched to *filename*.
.pi *program*		-	-	Pipe output to *program* (NROFF only). This request must occur *before* any printing occurs. No arguments are transmitted to *program*.

20. Miscellaneous

Request Form	*Initial Value*	*If No Argument*	*Notes*	*Explanation*
.mc *c N*	-	off	E,**m**	Specifies that a *margin* character *c* appear a distance *N* to the right of the right margin after each non-empty text line (except those produced by **tl**). If the output line is too-long (as can happen in nofill mode) the character will be appended to the

				line. If *N* is not given, the previous *N* is used; the initial *N* is 0.2 inches in NROFF and 1 em in TROFF. The margin character used with this paragraph was a 12-point box-rule.
.tm *string*	-	newline	-	After skipping initial blanks, *string* (rest of the line) is read in *copy mode* and written on the user's terminal.
.ig *yy*	-	*.yy*=**..**	-	Ignore input lines. **ig** behaves exactly like **de** (§7) except that the input is discarded. The input is read in *copy mode*, and any auto-incremented registers will be affected.
.pm *t*	-	all	-	Print macros. The names and sizes of all of the defined macros and strings are printed on the user's terminal; if *t* is given, only the total of the sizes is printed. The sizes is given in *blocks* of 128 characters.
.fl	-	-	B	Flush output buffer. Used in interactive debugging to force output.

21. Output and Error Messages.

The output from **tm**, **pm**, and the prompt from **rd**, as well as various *error* messages are written onto UNIX's *standard message* output. The latter is different from the *standard output*, where NROFF formatted output goes. By default, both are written onto the user's terminal, but they can be independently redirected.

Various *error* conditions may occur during the operation of NROFF and TROFF. Certain less serious errors having only local impact do not cause processing to terminate. Two examples are *word overflow*, caused by a word that is too large to fit into the word buffer (in fill mode), and *line overflow*, caused by an output line that grew too large to fit in the line buffer; in both cases, a message is printed, the offending excess is discarded, and the affected word or line is marked at the point of truncation with a ∗ in NROFF and a ☜ in TROFF. The philosophy is to continue processing, if possible, on the grounds that output useful for debugging may be produced. If a serious error occurs, processing terminates, and an appropriate message is printed. Examples are the inability to create, read, or write files, and the exceeding of certain internal limits that make future output unlikely to be useful.

TUTORIAL EXAMPLES

T1. Introduction

Although NROFF and TROFF have by design a syntax reminiscent of earlier text processors* with the intent of easing their use, it is almost always necessary to prepare at least a small set of macro definitions to describe most documents. Such common formatting needs as page margins and footnotes are deliberately not built into NROFF and TROFF. Instead, the macro and string definition, number register, diversion, environment switching, page-position trap, and conditional input mechanisms provide the basis for user-defined implementations.

The examples to be discussed are intended to be useful and somewhat realistic, but won't necessarily cover all relevant contingencies. Explicit numerical parameters are used in the examples to make them easier to read and to illustrate typical values. In many cases, number registers would really be used to reduce the number of places where numerical information is kept, and to concentrate conditional parameter initialization like that which depends on whether TROFF or NROFF is being used.

T2. Page Margins

As discussed in §3, *header* and *footer* macros are usually defined to describe the top and bottom page margin areas respectively. A trap is planted at page position 0 for the header, and at −*N* (*N* from the page bottom) for the footer. The simplest such definitions might be

```
.de hd          \"define header
´sp 1i
..              \"end definition
.de fo          \"define footer
´bp
..              \"end definition
.wh 0 hd
.wh −1i fo
```

which provide blank 1 inch top and bottom margins. The header will occur on the *first* page, only if the definition and trap exist prior to the initial pseudo-page transition (§3). In fill mode, the output line that springs the footer trap was typically forced out because some part or whole word didn't fit on it. If anything in the footer and header that follows causes a *break*, that word or part word will be forced out. In this and other examples, requests like **bp** and **sp** that normally cause breaks are invoked using the *no-break* control character ´ to avoid this. When the header/footer design contains material requiring independent text processing, the environment may be switched, avoiding most interaction with the running text.

A more realistic example would be

```
.de hd              \"header
.if t .tl ´\(rn´´\(rn´  \"troff cut mark
.if \\n%>1 \{\
´sp |0.5i−1         \"tl base at 0.5i
.tl ´´− % −´´       \"centered page number
.ps                 \"restore size
.ft                 \"restore font
.vs \}              \"restore vs
´sp |1.0i           \"space to 1.0i
.ns                 \"turn on no-space mode
..
.de fo              \"footer
.ps 10              \"set footer/header size
.ft R               \"set font
.vs 12p             \"set base-line spacing
.if \\n%=1 \{\
´sp |\\n(.pu−0.5i−1  \"tl base 0.5i up
.tl ´´− % −´´ \}    \"first page number
´bp
..
.wh 0 hd
.wh −1i fo
```

which sets the size, font, and base-line spacing for the header/footer material, and ultimately restores them. The material in this case is a page number at the bottom of the first page and at the top of the remaining pages. If TROFF is used, a *cut mark* is drawn in the form of *root-en*'s at each margin. The **sp**'s refer to absolute positions to avoid dependence on the base-line spacing. Another reason for this in the footer is that the footer is invoked by printing a line whose vertical spacing swept past the trap position by possibly as much as the base-line spacing. The *no-space* mode is turned on at the end of **hd** to render ineffective accidental occurrences of **sp** at the top of the running text.

---

*For example: P. A. Crisman, Ed., *The Compatible Time-Sharing System,* MIT Press, 1965, Section AH9.01 (Description of RUNOFF program on MIT's CTSS system).

The above method of restoring size, font, etc. presupposes that such requests (that set *previous* value) are *not* used in the running text. A better scheme is save and restore both the current *and* previous values as shown for size in the following:

```
.de fo
.nr s1 \\n(.s      \"current size
.ps
.nr s2 \\n(.s      \"previous size
. ---              \"rest of footer
..
.de hd
. ---              \"header stuff
.ps \\n(s2         \"restore previous size
.ps \\n(s1         \"restore current size
..
```

Page numbers may be printed in the bottom margin by a separate macro triggered during the footer's page ejection:

```
.de bn             \"bottom number
.tl ´´– % –´´      \"centered page number
..
.wh –0.5i–1v bn \"tl base 0.5i up
```

T3. Paragraphs and Headings

The housekeeping associated with starting a new paragraph should be collected in a paragraph macro that, for example, does the desired preparagraph spacing, forces the correct font, size, base-line spacing, and indent, checks that enough space remains for *more than one* line, and requests a temporary indent.

```
.de pg             \"paragraph
.br                \"break
.ft R              \"force font,
.ps 10             \"size,
.vs 12p            \"spacing,
.in 0              \"and indent
.sp 0.4            \"prespace
.ne 1+\\n(.Vu      \"want more than 1 line
.ti 0.2i           \"temp indent
..
```

The first break in **pg** will force out any previous partial lines, and must occur before the **vs**. The forcing of font, etc. is partly a defense against prior error and partly to permit things like section heading macros to set parameters only once. The prespacing parameter is suitable for TROFF; a larger space, at least as big as the output device vertical resolution, would be more suitable in NROFF. The choice of remaining space to test for in the **ne** is the smallest amount greater than one line (the **.V** is the available vertical resolution).

A macro to automatically number section headings might look like:

```
.de sc             \"section
. ---              \"force font, etc.
.sp 0.4            \"prespace
.ne 2.4+\\n(.Vu \"want 2.4+ lines
.fi
\\n+S.
..
.nr S 0 1          \"init S
```

The usage is **.sc**, followed by the section heading text, followed by **.pg**. The **ne** test value includes one line of heading, 0.4 line in the following **pg**, and one line of the paragraph text. A word consisting of the next section number and a period is produced to begin the heading line. The format of the number may be set by **af** (§8).

Another common form is the labeled, indented paragraph, where the label protrudes left into the indent space.

```
.de lp             \"labeled paragraph
.pg
.in 0.5i           \"paragraph indent
.ta 0.2i 0.5i      \"label, paragraph
.ti 0
\t\\$1\t\c         \"flow into paragraph
..
```

The intended usage is "**.lp** *label*"; *label* will begin at 0.2 inch, and cannot exceed a length of 0.3 inch without intruding into the paragraph. The label could be right adjusted against 0.4 inch by setting the tabs instead with **.ta 0.4iR 0.5i**. The last line of **lp** ends with **\c** so that it will become a part of the first line of the text that follows.

T4. Multiple Column Output

The production of multiple column pages requires the footer macro to decide whether it was invoked by other than the last column, so that it will begin a new column rather than produce the bottom margin. The header can initialize a column register that the footer will increment and test. The following is arranged for two columns, but is easily modified for more.

```
.de hd             \"header
. ---
.nr cl 0 1         \"init column count
.mk                \"mark top of text
..
.de fo             \"footer
.ie \\n+(cl<2 \{\
.po +3.4i          \"next column; 3.1+0.3
.rt                \"back to mark
```

```
.ns \}          \"no-space mode
.el \{\
.po \\nMu       \"restore left margin
.  ---
´bp \}
..
.ll 3.1i        \"column width
.nr M \\n(.o    \"save left margin
```

Typically a portion of the top of the first page contains full width text; the request for the narrower line length, as well as another **.mk** would be made where the two column output was to begin.

T5. Footnote Processing

The footnote mechanism to be described is used by imbedding the footnotes in the input text at the point of reference, demarcated by an initial **.fn** and a terminal **.ef**:

```
.fn
Footnote text and control lines...
.ef
```

In the following, footnotes are processed in a separate environment and diverted for later printing in the space immediately prior to the bottom margin. There is provision for the case where the last collected footnote doesn't completely fit in the available space.

```
.de hd          \"header
.  ---
.nr x 0 1       \"init footnote count
.nr y 0-\\nb    \"current footer place
.ch fo -\\nbu   \"reset footer trap
.if \\n(dn .fz  \"leftover footnote
..
.de fo          \"footer
.nr dn 0        \"zero last diversion size
.if \\nx \{\
.ev 1           \"expand footnotes in ev1
.nf             \"retain vertical size
.FN             \"footnotes
.rm FN          \"delete it
.if "\\n(.z"fy" .di  \"end overflow diversion
.nr x 0         \"disable fx
.ev  \}         \"pop environment
.  ---
´bp
..
.de fx          \"process footnote overflow
.if \\nx .di fy \"divert overflow
..
.de fn          \"start footnote
.da FN          \"divert (append) footnote
.ev 1           \"in environment 1
.if \\n+x=1 .fs  \"if first, include separator
.fi             \"fill mode
..
.de ef          \"end footnote
.br             \"finish output
.nr z \\n(.v    \"save spacing
.ev             \"pop ev
.di             \"end diversion
.nr y -\\n(dn   \"new footer position,
.if \\nx=1 .nr y -(\\n(.v-\\nz) \
                \"uncertainty correction
.ch fo \\nyu    \"y is negative
.if (\\n(nl+1v)>(\\n(.p+\\ny) \
.ch fo \\n(nlu+1v \"it didn't fit
..
.de fs          \"separator
\l´1i´          \"1 inch rule
.br
..
.de fz          \"get leftover footnote
.fn
.nf             \"retain vertical size
.fy             \"where fx put it
.ef
..
.nr b 1.0i      \"bottom margin size
.wh 0 hd        \"header trap
.wh 12i fo      \"footer trap, temp position
.wh -\\nbu fx   \"fx at footer position
.ch fo -\\nbu   \"conceal fx with fo
```

The header **hd** initializes a footnote count register **x**, and sets both the current footer trap position register **y** and the footer trap itself to a nominal position specified in register **b**. In addition, if the register **dn** indicates a leftover footnote, **fz** is invoked to reprocess it. The footnote start macro **fn** begins a diversion (append) in environment 1, and increments the count **x**; if the count is one, the footnote separator **fs** is interpolated. The separator is kept in a separate macro to permit user redefinition. The footnote end macro **ef** restores the previous environment and ends the diversion after saving the spacing size in register **z**. **y** is then decremented by the size of the footnote, available in **dn**; then on the first footnote, **y** is further decremented by the difference in vertical base-line spacings of the two environments, to prevent the late triggering the footer trap from causing the last line of the combined footnotes to overflow. The footer trap is then set to the lower (on the page) of **y** or the current page position (**nl**) plus one line, to allow for printing the reference line. If indicated by **x**, the footer **fo** rereads the footnotes from **FN** in nofill mode in environment 1, and deletes **FN**. If the footnotes were too large to fit, the macro **fx** will be trap-invoked to redivert the overflow into **fy**, and the register **dn** will later indicate to the header whether **fy** is empty. Both **fo** and **fx** are planted in the nominal footer trap position in an order that causes **fx** to be concealed unless the **fo** trap is moved. The footer then terminates the overflow diversion, if necessary, and zeros **x** to disable **fx**, because the uncertainty correction together with a not-too-late triggering of the footer can result in the footnote rereading finishing before reaching the **fx** trap.

A good exercise for the student is to combine the multiple-column and footnote mechanisms.

T6. The Last Page

After the last input file has ended, NROFF and TROFF invoke the *end macro* (§7), if any, and when it finishes, eject the remainder of the page. During the eject, any traps encountered are processed normally. At the *end* of this last page, processing terminates *unless* a partial line, word, or partial word remains. If it is desired that another page be started, the end-macro

```
.de en            \"end-macro
\c
´bp
..
.em en
```

will deposit a null partial word, and effect another last page.

Table I

Font Style Examples

The following fonts are printed in 12-point, with a vertical spacing of 14-point, and with non-alphanumeric characters separated by ¼ em space. The Special Mathematical Font was specially prepared for Bell Laboratories by Graphic Systems, Inc. of Hudson, New Hampshire. The Times Roman, Italic, and Bold are among the many standard fonts available from that company.

Times Roman

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
• □ — ‐ _ ¼ ½ ¾ fi fl ff ffi ffl ° † ′ ¢ ® ©

Times Italic

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
• □ — ‐ _ ¼ ½ ¾ *fi fl ff ffi ffl* ° † ′ ¢ ® ©

Times Bold

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + − . , / : ; = ? [ ] |
• ■ — ‐ _ ¼ ½ ¾ **fi fl ff ffi ffl** ° † ′ ¢ ® ©

Special Mathematical Font

" ´ \ ^ _ ` ˜ / < > { } # @ + − = ∗
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ ς τ υ φ χ ψ ω
Γ Δ Θ Λ Ξ Π Σ Υ Φ Ψ Ω
√ ‾ ≥ ≤ ≡ ~ ≃ ≠ → ← ↑ ↓ × ÷ ± ∪ ∩ ⊂ ⊃ ⊆ ⊇ ∞ ∂
§ ∇ ¬ ∫ ∝ ∅ ∈ ‡ ☞ ☜ 🔔 | ○ ⎧ ⎩ ⎫ ⎭ ⎨ ⎬ | ⌊ ⌋ ⌈ ⌉ |

Table II

Input Naming Conventions for ´, `,and –
and for Non-ASCII Special Characters

Non-ASCII characters and *minus* on the standard fonts.

Char	Input Name	Character Name	Char	Input Name	Character Name
’	´	close quote	ﬁ	\(fi	ﬁ
‘	`	open quote	ﬂ	\(fl	ﬂ
—	\(em	3⁄4 Em dash	ﬀ	\(ff	ﬀ
-	–	hyphen or	ﬃ	\(Fi	ﬃ
-	\(hy	hyphen	ﬄ	\(Fl	ﬄ
–	\–	current font minus	°	\(de	degree
•	\(bu	bullet	†	\(dg	dagger
□	\(sq	square	′	\(fm	foot mark
_	\(ru	rule	¢	\(ct	cent sign
¼	\(14	1⁄4	®	\(rg	registered
½	\(12	1⁄2	©	\(co	copyright
¾	\(34	3⁄4			

Non-ASCII characters and ´, `, _, +, −, =, and ∗ on the special font.

The ASCII characters @, #, ", ´, `, <, >, \, {, }, ˜, ˆ, and _ exist *only* on the special font and are printed as a 1-em space if that font is not mounted. The following characters exist only on the special font except for the upper case Greek letter names followed by † which are mapped into upper case English letters in whatever font is mounted on font position one (default Times Roman). The special math plus, minus, and equals are provided to insulate the appearance of equations from the choice of standard fonts.

Char	Input Name	Character Name	Char	Input Name	Character Name
+	\(pl	math plus	λ	\(*l	lambda
−	\(mi	math minus	μ	\(*m	mu
=	\(eq	math equals	ν	\(*n	nu
∗	\(**	math star	ξ	\(*c	xi
§	\(sc	section	ο	\(*o	omicron
´	\(aa	acute accent	π	\(*p	pi
`	\(ga	grave accent	ρ	\(*r	rho
_	\(ul	underrule	σ	\(*s	sigma
⁄	\(sl	slash (matching backslash)	ς	\(ts	terminal sigma
α	\(*a	alpha	τ	\(*t	tau
β	\(*b	beta	υ	\(*u	upsilon
γ	\(*g	gamma	φ	\(*f	phi
δ	\(*d	delta	χ	\(*x	chi
ε	\(*e	epsilon	ψ	\(*q	psi
ζ	\(*z	zeta	ω	\(*w	omega
η	\(*y	eta	Α	\(*A	Alpha†
θ	\(*h	theta	Β	\(*B	Beta†
ι	\(*i	iota	Γ	\(*G	Gamma
κ	\(*k	kappa	Δ	\(*D	Delta

Char	*Input Name*	*Character Name*
Ε	\(*E	Epsilon†
Ζ	\(*Z	Zeta†
Η	\(*Y	Eta†
Θ	\(*H	Theta
Ι	\(*I	Iota†
Κ	\(*K	Kappa†
Λ	\(*L	Lambda
Μ	\(*M	Mu†
Ν	\(*N	Nu†
Ξ	\(*C	Xi
Ο	\(*O	Omicron†
Π	\(*P	Pi
Ρ	\(*R	Rho†
Σ	\(*S	Sigma
Τ	\(*T	Tau†
Υ	\(*U	Upsilon
Φ	\(*F	Phi
Χ	\(*X	Chi†
Ψ	\(*Q	Psi
Ω	\(*W	Omega
√	\(sr	square root
‾	\(rn	root en extender
≥	\(>=	>=
≤	\(<=	<=
≡	\(==	identically equal
≃	\(~=	approx =
~	\(ap	approximates
≠	\(!=	not equal
→	\(->	right arrow
←	\(<-	left arrow
↑	\(ua	up arrow
↓	\(da	down arrow
×	\(mu	multiply
÷	\(di	divide
±	\(+-	plus-minus
∪	\(cu	cup (union)
∩	\(ca	cap (intersection)
⊂	\(sb	subset of
⊃	\(sp	superset of
⊆	\(ib	improper subset
⊇	\(ip	improper superset
∞	\(if	infinity
∂	\(pd	partial derivative
∇	\(gr	gradient
¬	\(no	not
∫	\(is	integral sign
∝	\(pt	proportional to
∅	\(es	empty set
∈	\(mo	member of
│	\(br	box vertical rule
‡	\(dd	double dagger

Char	*Input Name*	*Character Name*
☞	\(rh	right hand
☜	\(lh	left hand
	\(bs	Bell System logo
│	\(or	or
○	\(ci	circle
⎧	\(lt	left top of big curly bracket
⎩	\(lb	left bottom
⎫	\(rt	right top
⎭	\(rb	right bot
⎨	\(lk	left center of big curly bracket
⎬	\(rk	right center of big curly bracket
│	\(bv	bold vertical
⌊	\(lf	left floor (left bottom of big square bracket)
⌋	\(rf	right floor (right bottom)
⌈	\(lc	left ceiling (left top)
⌉	\(rc	right ceiling (right top)

Summary of Changes to N/TROFF Since October 1976 Manual

Options

-h (Nroff only) Output tabs used during horizontal spacing to speed output as well as reduce output byte count. Device tab settings assumed to be every 8 nominal character widths. The default settings of input (logical) tabs is also initialized to every 8 nominal character widths.

-z Efficiently suppresses formatted output. Only message output will occur (from "tm"s and diagnostics).

Old Requests

.ad c The adjustment type indicator "c" may now also be a number previously obtained from the ".j" register (see below).

.so name The contents of file "name" will be interpolated at the point the "so" is encountered. Previously, the interpolation was done upon return to the file-reading input level.

New Request

.ab text Prints "text" on the message output and terminates without further processing. If "text" is missing, "User Abort." is printed. Does not cause a break. The output buffer is flushed.

.fz F N forces font "F" to be in size N. N may have the form N, +N, or -N. For example,

    .fz 3 -2

will cause an implicit \s-2 every time font 3 is entered, and a corresponding \s+2 when it is left. Special font characters occurring during the reign of font F will have the same size modification. If special characters are to be treated differently,

    .fz S F N

may be used to specify the size treatment of special characters during font F. For example,

    .fz 3 -3
    .fz S 3 -0

will cause automatic reduction of font 3 by 3 points while the special characters would not be affected. Any ‘‘.fp’’ request specifying a font on some position must precede ‘‘.fz’’ requests relating to that position.

New Predefined Number Registers.

.k Read-only. Contains the horizontal size of the text portion (without indent) of the current partially collected output line, if any, in the current environment.

.j Read-only. A number representing the current adjustment mode and type. Can be saved and later given to the "ad" request to restore a previous mode.

.P Read-only. 1 if the current page is being printed, and zero otherwise.

.L Read-only. Contains the current line-spacing parameter ("ls").

c. General register access to the input line-number in the current input file. Contains the same value as the read-only ".c" register.

A TROFF Tutorial

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

troff is a text-formatting program for driving the Graphic Systems phototypesetter on the UNIX† and GCOS operating systems. This device is capable of producing high quality text; this paper is an example of **troff** output.

The phototypesetter itself normally runs with four fonts, containing roman, italic and bold letters (as on this page), a full greek alphabet, and a substantial number of special characters and mathematical symbols. Characters can be printed in a range of sizes, and placed anywhere on the page.

troff allows the user full control over fonts, sizes, and character positions, as well as the usual features of a formatter — right-margin justification, automatic hyphenation, page titling and numbering, and so on. It also provides macros, arithmetic variables and operations, and conditional testing, for complicated formatting tasks.

This document is an introduction to the most basic use of **troff**. It presents just enough information to enable the user to do simple formatting tasks like making viewgraphs, and to make incremental changes to existing packages of **troff** commands. In most respects, the UNIX formatter **nroff** is identical to **troff**, so this document also serves as a tutorial on **nroff**.

August 4, 1978

†UNIX is a Trademark of Bell Laboratories.

A TROFF Tutorial

Brian W. Kernighan

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

troff [1] is a text-formatting program, written by J. F. Ossanna, for producing high-quality printed output from the phototypesetter on the UNIX and GCOS operating systems. This document is an example of **troff** output.

The single most important rule of using **troff** is not to use it directly, but through some intermediary. In many ways, **troff** resembles an assembly language — a remarkably powerful and flexible one — but nonetheless such that many operations must be specified at a level of detail and in a form that is too hard for most people to use effectively.

For two special applications, there are programs that provide an interface to **troff** for the majority of users. **eqn** [2] provides an easy to learn language for typesetting mathematics; the **eqn** user need know no **troff** whatsoever to typeset mathematics. **tbl** [3] provides the same convenience for producing tables of arbitrary complexity.

For producing straight text (which may well contain mathematics or tables), there are a number of 'macro packages' that define formatting rules and operations for specific styles of documents, and reduce the amount of direct contact with **troff**. In particular, the '–ms' [4] and PWB/MM [5] packages for Bell Labs internal memoranda and external papers provide most of the facilities needed for a wide range of document preparation. (This memo was prepared with '–ms'.) There are also packages for viewgraphs, for simulating the older **roff** formatters on UNIX and GCOS, and for other special applications. Typically you will find these packages easier to use than **troff** once you get beyond the most trivial operations; you should always consider them first.

In the few cases where existing packages don't do the whole job, the solution is *not* to write an entirely new set of **troff** instructions from scratch, but to make small changes to adapt packages that already exist.

In accordance with this philosophy of letting someone else do the work, the part of **troff** described here is only a small part of the whole, although it tries to concentrate on the more useful parts. In any case, there is no attempt to be complete. Rather, the emphasis is on showing how to do simple things, and how to make incremental changes to what already exists. The contents of the remaining sections are:

2. Point sizes and line spacing
3. Fonts and special characters
4. Indents and line length
5. Tabs
6. Local motions: Drawing lines and characters
7. Strings
8. Introduction to macros
9. Titles, pages and numbering
10. Number registers and arithmetic
11. Macros with arguments
12. Conditionals
13. Environments
14. Diversions

Appendix: Typesetter character set

The **troff** described here is the C-language version running on UNIX at Murray Hill, as documented in [1].

To use **troff** you have to prepare not only the actual text you want printed, but some information that tells *how* you want it printed. (Readers who use **roff** will find the approach familiar.) For **troff** the text and the formatting information are often intertwined quite intimately. Most commands to **troff** are placed on a line separate from the text itself, beginning with a period (one command per line). For example,

```
Some text.
.ps 14
Some more text.
```

will change the 'point size', that is, the size of the letters being printed, to '14 point' (one point is 1/72 inch) like this:

Some text. Some more text.

Occasionally, though, something special occurs in the middle of a line — to produce

Area = πr^2

you have to type

Area = \(*p\fIr\fR\ |\s8\u2\d\s0

(which we will explain shortly). The backslash character \ is used to introduce **troff** commands and special characters within a line of text.

2. Point Sizes; Line Spacing

As mentioned above, the command .ps sets the point size. One point is 1/72 inch, so 6-point characters are at most 1/12 inch high, and 36-point characters are ½ inch. There are 15 point sizes, listed below.

6 point: Pack my box with five dozen liquor jugs.
7 point: Pack my box with five dozen liquor jugs.
8 point: Pack my box with five dozen liquor jugs.
9 point: Pack my box with five dozen liquor jugs.
10 point: Pack my box with five dozen liquor
11 point: Pack my box with five dozen
12 point: Pack my box with five dozen
14 point: Pack my box with five
16 point 18 point 20 point
22 24 28 36

If the number after .ps is not one of these legal sizes, it is rounded up to the next valid value, with a maximum of 36. If no number follows .ps, **troff** reverts to the previous size, whatever it was. **troff** begins with point size 10, which is usually fine. This document is in 9 point.

The point size can also be changed in the middle of a line or even a word with the in-line command \s. To produce

UNIX runs on a PDP-11/45

type

\s8UNIX\s10 runs on a \s8PDP-\s1011/45

As above, \s should be followed by a legal point size, except that \s0 causes the size to revert to its previous value. Notice that \s1011 can be understood correctly as ‘size 10, followed by an 11’, if the size is legal, but not otherwise. Be cautious with similar constructions.

Relative size changes are also legal and useful:

\s−2UNIX\s+2

temporarily decreases the size, whatever it is, by two points, then restores it. Relative size changes have the advantage that the size difference is independent of the starting size of the document. The amount of the relative change is restricted to a single digit.

The other parameter that determines what the type looks like is the spacing between lines, which is set independently of the point size. Vertical spacing is measured from the bottom of one line to the bottom of the next. The command to control vertical spacing is .vs. For running text, it is usually best to set the vertical spacing about 20% bigger than the character size. For example, so far in this document, we have used ‘‘9 on 11’’, that is,

```
.ps 9
.vs 11p
```

If we changed to

```
.ps 9
.vs 9p
```

the running text would look like this. After a few lines, you will agree it looks a little cramped. The right vertical spacing is partly a matter of taste, depending on how much text you want to squeeze into a given space, and partly a matter of traditional printing style. By default, **troff** uses 10 on 12.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. This is 12 on 14.

Point size and vertical spacing make a substantial difference in the amount of text per square inch. For example, 10 on 12 uses about twice as much space as 7 on 8. This is 6 on 7, which is even smaller. It packs a lot more words per line, but you can go blind trying to read it.

When used without arguments, .ps and .vs revert to the previous size and vertical spacing respectively.

The command .sp is used to get extra vertical space. Unadorned, it gives you one extra blank line (one .vs, whatever that has been set to). Typically, that’s more or less than you want, so .sp can be followed by information about how much space you want —

```
.sp 2i
```

means ‘two inches of vertical space’.

```
.sp 2p
```

means ‘two points of vertical space’; and

```
.sp 2
```

means ‘two vertical spaces’ — two of whatever .vs is set to (this can also be made explicit with .sp 2v); **troff** also understands decimal fractions in most places, so

```
.sp 1.5i
```

is a space of 1.5 inches. These same scale factors can be used after .vs to define line spacing, and in fact after most commands that deal with physical dimensions.

It should be noted that all size numbers are converted internally to 'machine units', which are 1/432 inch (1/6 point). For most purposes, this is enough resolution that you don't have to worry about the accuracy of the representation. The situation is not quite so good vertically, where resolution is 1/144 inch (1/2 point).

3. Fonts and Special Characters

troff and the typesetter allow four different fonts at any one time. Normally three fonts (Times roman, italic and bold) and one collection of special characters are permanently mounted.

abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz 0123456789
ABCDEFGHIJKLMNOPQRSTUVWXYZ

The greek, mathematical symbols and miscellany of the special font are listed in Appendix A.

troff prints in roman unless told otherwise. To switch into bold, use the .ft command

.ft B

and for italics,

.ft I

To return to roman, use .ft R; to return to the previous font, whatever it was, use either .ft P or just .ft. The 'underline' command

.ul

causes the next input line to print in italics. .ul can be followed by a count to indicate that more than one line is to be italicized.

Fonts can also be changed within a line or word with the in-line command \f:

bold*face* text

is produced by

\fBbold\fIface\fR text

If you want to do this so the previous font, whatever it was, is left undisturbed, insert extra \fP commands, like this:

\fBbold\fP\fIface\fP\fR text\fP

Because only the immediately previous font is remembered, you have to restore the previous font after each change or you can lose it. The same is true of .ps and .vs when used without an argument.

There are other fonts available besides the standard set, although you can still use only four at any given time. The command .fp tells **troff** what fonts are physically mounted on the typesetter:

.fp 3 H

says that the Helvetica font is mounted on position 3. (For a complete list of fonts and what they look like, see the **troff** manual.) Appropriate .fp commands should appear at the beginning of your document if you do not use the standard fonts.

It is possible to make a document relatively independent of the actual fonts used to print it by using font numbers instead of names; for example, \f3 and .ft˜3 mean 'whatever font is mounted at position 3', and thus work for any setting. Normal settings are roman font on 1, italic on 2, bold on 3, and special on 4.

There is also a way to get 'synthetic' bold fonts by overstriking letters with a slight offset. Look at the .bd command in [1].

Special characters have four-character names beginning with \(, and they may be inserted anywhere. For example,

¼ + ½ = ¾

is produced by

\(14 + \(12 = \(34

In particular, greek letters are all of the form \(*−, where − is an upper or lower case roman letter reminiscent of the greek. Thus to get

$\Sigma(\alpha\times\beta) \rightarrow \infty$

in bare **troff** we have to type

\(*S(\(*a\(mu\(*b) \(−> \(if

That line is unscrambled as follows:

\(*S	Σ
(	(
\(*a	α
\(mu	$\times$
\(*b	β
)	)
\(−>	$\rightarrow$
\(if	∞

A complete list of these special names occurs in Appendix A.

In **eqn** [2] the same effect can be achieved with the input

SIGMA (alpha times beta) −> inf

which is less concise, but clearer to the uninitiated.

Notice that each four-character name is a single character as far as **troff** is concerned — the 'translate' command

.tr \(mi\(em

is perfectly clear, meaning

```
.tr –—
```

that is, to translate – into —.

Some characters are automatically translated into others: grave ` and acute ´ accents (apostrophes) become open and close single quotes ‘~’; the combination of ‘‘...’’ is generally preferable to the double quotes "...". Similarly a typed minus sign becomes a hyphen -. To print an explicit – sign, use \-. To get a backslash printed, use \e.

4. Indents and Line Lengths

troff starts with a line length of 6.5 inches, too wide for 8½×11 paper. To reset the line length, use the .ll command, as in

```
.ll 6i
```

As with .sp, the actual length can be specified in several ways; inches are probably the most intuitive.

The maximum line length provided by the typesetter is 7.5 inches, by the way. To use the full width, you will have to reset the default physical left margin (‘‘page offset’’), which is normally slightly less than one inch from the left edge of the paper. This is done by the .po command.

```
.po 0
```

sets the offset as far to the left as it will go.

The indent command .in causes the left margin to be indented by some specified amount from the page offset. If we use .in to move the left margin in, and .ll to move the right margin to the left, we can make offset blocks of text:

```
.in 0.3i
.ll –0.3i
text to be set into a block
.ll +0.3i
.in –0.3i
```

will create a block that looks like this:

> Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Notice the use of ‘+’ and ‘–’ to specify the amount of change. These change the previous setting by the specified amount, rather than just overriding it. The distinction is quite important: .ll +1i makes lines one inch longer; .ll 1i makes them one inch *long*.

With .in, .ll and .po, the previous value is used if no argument is specified.

To indent a single line, use the ‘temporary indent’ command .ti. For example, all paragraphs in this memo effectively begin with the command

```
.ti 3
```

Three of what? The default unit for .ti, as for most horizontally oriented commands (.ll, .in, .po), is ems; an em is roughly the width of the letter ‘m’ in the current point size. (Precisely, a em in size *p* is *p* points.) Although inches are usually clearer than ems to people who don’t set type for a living, ems have a place: they are a measure of size that is proportional to the current point size. If you want to make text that keeps its proportions regardless of point size, you should use ems for all dimensions. Ems can be specified as scale factors directly, as in .ti 2.5m.

Lines can also be indented negatively if the indent is already positive:

```
.ti –0.3i
```

causes the next line to be moved back three tenths of an inch. Thus to make a decorative initial capital, we indent the whole paragraph, then move the letter ‘P’ back with a .ti command:

> Pater noster qui est in caelis sanctificetur nomen tuum; adveniat regnum tuum; fiat voluntas tua, sicut in caelo, et in terra. ... Amen.

Of course, there is also some trickery to make the ‘P’ bigger (just a ‘\s36P\s0’), and to move it down from its normal position (see the section on local motions).

5. Tabs

Tabs (the ASCII ‘horizontal tab’ character) can be used to produce output in columns, or to set the horizontal position of output. Typically tabs are used only in unfilled text. Tab stops are set by default every half inch from the current indent, but can be changed by the .ta command. To set stops every inch, for example,

```
.ta 1i 2i 3i 4i 5i 6i
```

Unfortunately the stops are left-justified only (as on a typewriter), so lining up columns of right-justified numbers can be painful. If you have many numbers, or if you need more complicated table layout, *don’t* use **troff** directly; use the **tbl** program described in [3].

For a handful of numeric columns, you can do it this way: Precede every number by enough blanks to make it line up when typed.

```
.nf
.ta 1i 2i 3i
  1 tab   2 tab   3
 40 tab  50 tab  60
700 tab 800 tab 900
.fi
```

Then change each leading blank into the string \0. This is a character that does not print, but that has the same width as a digit. When printed, this will produce

1	2	3
40	50	60
700	800	900

It is also possible to fill up tabbed-over space with some character other than blanks by setting the 'tab replacement character' with the .tc command:

```
.ta 1.5i 2.5i
.tc \(ru          (\(ru is "_")
```
Name *tab* Age *tab*

produces

Name___________________ Age ___________

To reset the tab replacement character to a blank, use **.tc** with no argument. (Lines can also be drawn with the **\l** command, described in Section 6.)

troff also provides a very general mechanism called 'fields' for setting up complicated columns. (This is used by **tbl**). We will not go into it in this paper.

6. Local Motions: Drawing lines and characters

Remember 'Area = πr^2' and the big 'P' in the Paternoster. How are they done? **troff** provides a host of commands for placing characters of any size at any place. You can use them to draw special characters or to tune your output for a particular appearance. Most of these commands are straightforward, but messy to read and tough to type correctly.

If you won't use **eqn**, subscripts and superscripts are most easily done with the half-line local motions **\u** and **\d**. To go back up the page half a point-size, insert a **\u** at the desired place; to go down, insert a **\d**. (**\u** and **\d** should always be used in pairs, as explained below.) Thus

```
Area = \(*pr\u2\d
```

produces

Area = πr^2

To make the '2' smaller, bracket it with **\s−2...\s0**. Since **\u** and **\d** refer to the current point size, be sure to put them either both inside or both outside the size changes, or you will get an unbalanced vertical motion.

Sometimes the space given by **\u** and **\d** isn't the right amount. The **\v** command can be used to request an arbitrary amount of vertical motion. The in-line command

```
\v´(amount)´
```

causes motion up or down the page by the amount specified in '(amount)'. For example, to move the 'P' down, we used

```
.in +0.6i            (move paragraph in)
.ll −0.3i            (shorten lines)
.ti −0.3i            (move P back)
\v´2´\s36P\s0\v´−2´ater noster qui est
in caelis ...
```

A minus sign causes upward motion, while no sign or a plus sign means down the page. Thus **\v′−2′** causes an upward vertical motion of two line spaces.

There are many other ways to specify the amount of motion —

```
\v´0.1i´
\v´3p´
\v´−0.5m´
```

and so on are all legal. Notice that the scale specifier **i** or **p** or **m** goes inside the quotes. Any character can be used in place of the quotes; this is also true of all other **troff** commands described in this section.

Since **troff** does not take within-the-line vertical motions into account when figuring out where it is on the page, output lines can have unexpected positions if the left and right ends aren't at the same vertical position. Thus **\v**, like **\u** and **\d**, should always balance upward vertical motion in a line with the same amount in the downward direction.

Arbitrary horizontal motions are also available — **\h** is quite analogous to **\v**, except that the default scale factor is ems instead of line spaces. As an example,

```
\h´−0.1i´
```

causes a backwards motion of a tenth of an inch. As a practical matter, consider printing the mathematical symbol '>>'. The default spacing is too wide, so **eqn** replaces this by

```
>\h´−0.3m´>
```

to produce ≫.

Frequently **\h** is used with the 'width function' **\w** to generate motions equal to the width of some character string. The construction

```
\w´thing´
```

is a number equal to the width of 'thing' in machine units (1/432 inch). All **troff** computations are ultimately done in these units. To move horizontally the width of an 'x', we can say

```
\h´\w´x´u´
```

As we mentioned above, the default scale factor for all horizontal dimensions is **m**, ems, so here we must have the **u** for machine units, or the motion produced will be far too large. **troff** is quite happy with the nested quotes, by the way, so long as you don't leave any out.

As a live example of this kind of construction, all of the command names in the text, like **.sp**, were done by overstriking with a slight offset. The commands for **.sp** are

```
.sp\h´−\w´.sp´u´\h´1u´.sp
```

That is, put out '.sp', move left by the width of '.sp', move right 1 unit, and print '.sp' again. (Of course there is a way to avoid typing that much input for each command name, which we will discuss in Section 11.)

There are also several special-purpose **troff** commands for local motion. We have already seen **\0**, which is an unpaddable white space of the same width as a digit. 'Unpaddable' means that it will never be widened or split across a line by line justification and filling. There is also \(blank), which is an unpaddable character the width of a space, \|, which is half that width, \^, which is one quarter of the width of a space, and **\&**, which has zero width. (This last one is useful, for example, in entering a text line which would otherwise begin with a '.'.)

The command **\o**, used like

```
\o´set of characters´
```

causes (up to 9) characters to be overstruck, centered on the widest. This is nice for accents, as in

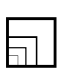
```
syst\o"e\(ga"me t\o"e\(aa"l\o"e\(aa"phonique
```

which makes

système téléphonique

The accents are **\(ga** and **\(aa**, or \` and \´; remember that each is just one character to **troff**.

You can make your own overstrikes with another special convention, **\z**, the zero-motion command. **\zx** suppresses the normal horizontal motion after printing the single character **x**, so another character can be laid on top of it. Although sizes can be changed within **\o**, it centers the characters on the widest, and there can be no horizontal or vertical motions, so **\z** may be the only way to get what you want:



is produced by

```
.sp 2
\s8\z\(sq\s14\z\(sq\s22\z\(sq\s36\(sq
```

The **.sp** is needed to leave room for the result.

As another example, an extra-heavy semicolon that looks like

; instead of ; or ;

can be constructed with a big comma and a big period above it:

```
\s+6\z,\v´−0.25m´.\v´0.25m´\s0
```

'0.25m' is an empirical constant.

A more ornate overstrike is given by the bracketing function **\b**, which piles up characters vertically, centered on the current baseline. Thus we can get big brackets, constructing them with piled-up smaller pieces:

$$\left\{ \left[\; x \; \right] \right\}$$

by typing in only this:

```
.sp
\b′\(lt\(lk\(lb′ \b′\(lc\(lf′ x \b′\(rc\(rf′ \b′\(rt\(rk\(rb′
```

troff also provides a convenient facility for drawing horizontal and vertical lines of arbitrary length with arbitrary characters. **\l′1i′** draws a line one inch long, like this: ________________. The length can be followed by the character to use if the _ isn't appropriate; **\l′0.5i.′** draws a half-inch line of dots: .... The construction **\L** is entirely analogous, except that it draws a vertical line instead of horizontal.

7. Strings

Obviously if a paper contains a large number of occurrences of an acute accent over a letter 'e', typing \o"e\´" for each é would be a great nuisance.

Fortunately, **troff** provides a way in which you can store an arbitrary collection of text in a 'string', and thereafter use the string name as a shorthand for its contents. Strings are one of several **troff** mechanisms whose judicious use lets you type a document with less effort and organize it so that extensive format changes can be made with few editing changes.

A reference to a string is replaced by whatever text the string was defined as. Strings are defined with the command **.ds**. The line

```
.ds e \o"e\´"
```

defines the string **e** to have the value **\o"e\´"**

String names may be either one or two characters long, and are referred to by ***x** for one character names or ***(xy** for two character names. Thus to get téléphone, given the definition of the string **e** as above, we can say t*el*ephone.

If a string must begin with blanks, define it as

```
.ds xx "      text
```

The double quote signals the beginning of the definition. There is no trailing quote; the end of the line terminates the string.

A string may actually be several lines long; if **troff** encounters a \ at the end of *any* line, it is thrown away and the next line added to the current one. So you can make a long string simply by ending each line but the last with a backslash:

```
.ds xx this \
is a very \
long string
```

Strings may be defined in terms of other strings, or even in terms of themselves; we will discuss some of these possibilities later.

8. Introduction to Macros

Before we can go much further in **troff**, we need to learn a bit about the macro facility. In its simplest form, a macro is just a shorthand notation quite similar to a string. Suppose we want every paragraph to start in exactly the same way — with a space and a temporary indent of two ems:

```
.sp
.ti +2m
```

Then to save typing, we would like to collapse these into one shorthand line, a **troff** 'command' like

```
.PP
```

that would be treated by **troff** exactly as

```
.sp
.ti +2m
```

.PP is called a *macro.* The way we tell **troff** what **.PP** means is to *define* it with the **.de** command:

```
.de PP
.sp
.ti +2m
..
```

The first line names the macro (we used '**.PP**' for 'paragraph', and upper case so it wouldn't conflict with any name that **troff** might already know about). The last line .. marks the end of the definition. In between is the text, which is simply inserted whenever **troff** sees the 'command' or macro call

```
.PP
```

A macro can contain any mixture of text and formatting commands.

The definition of **.PP** has to precede its first use; undefined macros are simply ignored. Names are restricted to one or two characters.

Using macros for commonly occurring sequences of commands is critically important. Not only does it save typing, but it makes later changes much easier. Suppose we decide that the paragraph indent is too small, the vertical space is much too big, and roman font should be forced. Instead of changing the whole document, we need only change the definition of **.PP** to something like

```
.de PP      \" paragraph macro
.sp 2p
.ti +3m
.ft R
..
```

and the change takes effect everywhere we used **.PP**.

\" is a **troff** command that causes the rest of the line to be ignored. We use it here to add comments to the macro definition (a wise idea once definitions get complicated).

As another example of macros, consider these two which start and end a block of offset, unfilled text, like most of the examples in this paper:

```
.de BS      \" start indented block
.sp
.nf
.in +0.3i
..
.de BE      \" end indented block
.sp
.fi
.in −0.3i
..
```

Now we can surround text like

```
Copy to
John Doe
Richard Roberts
Stanley Smith
```

by the commands **.BS** and **.BE**, and it will come out as it did above. Notice that we indented by **.in +0.3i** instead of **.in 0.3i**. This way we can nest our uses of **.BS** and **BE** to get blocks within blocks.

If later on we decide that the indent should be 0.5i, then it is only necessary to change the definitions of **.BS** and **.BE**, not the whole paper.

9. Titles, Pages and Numbering

This is an area where things get tougher, because nothing is done for you automatically. Of necessity, some of this section is a cookbook, to be copied literally until you get some experience.

Suppose you want a title at the top of each page, saying just

```
~~~~left top                center top                right top~~~~
```

In **roff**, one can say

```
.he ´left top´center top´right top´
.fo ´left bottom´center bottom´right bottom´
```

to get headers and footers automatically on every page. Alas, this doesn't work in **troff**, a serious hardship for the novice. Instead you have to do a lot of specification.

You have to say what the actual title is (easy); when to print it (easy enough); and what to do at and around the title line (harder). Taking these in reverse order, first we define a macro **.NP** (for 'new page') to process titles and the like at the end of one page and the beginning of the next:

```
.de NP
′bp
′sp 0.5i
.tl ´left top´center top´right top´
′sp 0.3i
..
```

To make sure we're at the top of a page, we issue a 'begin page' command **′bp**, which causes a skip to top-of-page (we'll explain the **′** shortly). Then we space down half an inch, print the title (the use of **.tl** should be self explanatory; later we will discuss parameterizing the titles), space another 0.3 inches, and we're done.

To ask for **.NP** at the bottom of each page, we have to say something like 'when the text is within an inch of the bottom of the page, start the processing for a new page.' This is done with a 'when' command **.wh**:

```
.wh  −1i  NP
```

(No '.' is used before NP; this is simply the name of a macro, not a macro call.) The minus sign means 'measure up from the bottom of the page', so '−1i' means 'one inch from the bottom'.

The **.wh** command appears in the input outside the definition of **.NP**; typically the input would be

```
.de NP
...
..
.wh −1i NP
```

Now what happens? As text is actually being output, **troff** keeps track of its vertical position on the page, and after a line is printed within one inch from the bottom, the **.NP** macro is activated. (In the jargon, the **.wh** command sets a *trap* at the specified place, which is 'sprung' when that point is passed.) **.NP** causes a skip to the top of the next page (that's what the **′bp** was for), then prints the title with the appropriate margins.

Why **′bp** and **′sp** instead of **.bp** and **.sp**? The answer is that **.sp** and **.bp**, like several other commands, cause a *break* to take place. That is, all the input text collected but not yet printed is flushed out as soon as possible, and the next input line is guaranteed to start a new line of output. If we had used **.sp** or **.bp** in the **.NP** macro, this would cause a break in the middle of the current output line when a new page is started. The effect would be to print the left-over part of that line at the top of the page, followed by the next input line on a new output line. This is *not* what we want. Using **′** instead of **.** for a command tells **troff** that no break is to take place — the output line currently being filled should *not* be forced out before the space or new page.

The list of commands that cause a break is short and natural:

```
.bp   .br   .ce   .fi   .nf   .sp   .in   .ti
```

All others cause *no* break, regardless of whether you use a **.** or a **′**. If you really need a break, add a **.br** command at the appropriate place.

One other thing to beware of — if you're changing fonts or point sizes a lot, you may find that if you cross a page boundary in an unexpected font or size, your titles come out in that size and font instead of what you intended. Furthermore, the length of a title is independent of the current line length, so titles will come out at the default length of 6.5 inches unless you change it, which is done with the **.lt** command.

There are several ways to fix the problems of point sizes and fonts in titles. For the simplest applications, we can change **.NP** to set the proper size and font for the title, then restore the previous values, like this:

```
.de NP
′bp
′sp 0.5i
.ft R           \" set title font to roman
.ps 10          \" and size to 10 point
.lt 6i          \" and length to 6 inches
.tl ´left´center´right´
.ps             \" revert to previous size
.ft P           \" and to previous font
′sp 0.3i
..
```

This version of **.NP** does *not* work if the fields in the **.tl** command contain size or font changes. To cope with that requires **troff**'s 'environment' mechanism, which we will discuss in Section 13.

To get a footer at the bottom of a page, you can modify **.NP** so it does some processing before the **′bp** command, or split the job into a footer macro invoked at the bottom margin and a header macro invoked at the top of the page. These variations are left as exercises.

Output page numbers are computed automatically as each page is produced (starting at 1), but no numbers are printed unless you ask for them explicitly. To get page numbers printed, include the character **%** in the **.tl** line at the position where you want the number to appear. For example

```
.tl ´´- % -´´
```

centers the page number inside hyphens, as on this page. You can set the page number at any time with either **.bp n**, which immediately starts a new page numbered **n**, or with **.pn n**, which sets the page number for the next page but doesn't cause a skip to the new page. Again, **.bp +n** sets the page number to **n** more than its current value; **.bp** means **.bp +1**.

10. Number Registers and Arithmetic

troff has a facility for doing arithmetic, and for defining and using variables with numeric values, called *number registers.* Number registers, like strings and macros, can be useful in setting up a document so it is easy to change later. And of course they serve for any sort of arithmetic computation.

Like strings, number registers have one or two character names. They are set by the **.nr** command, and are referenced anywhere by **\nx** (one character name) or **\n(xy** (two character name).

There are quite a few pre-defined number registers maintained by **troff**, among them **%** for the current page number; **nl** for the current vertical position on the page; **dy**, **mo** and **yr** for the current day, month and year; and **.s** and **.f** for the current size and font. (The font is a number from 1 to 4.) Any of these can be used in computations like any other register, but some, like **.s** and **.f**, cannot be changed with **.nr**.

As an example of the use of number registers, in the −**ms** macro package [4], most significant parameters are defined in terms of the values of a handful of number registers. These include the point size for text, the vertical spacing, and the line and title lengths. To set the point size and vertical spacing for the following paragraphs, for example, a user may say

```
.nr PS 9
.nr VS 11
```

The paragraph macro **.PP** is defined (roughly) as follows:

```
.de PP
.ps \\n(PS        \" reset size
.vs \\n(VSp       \" spacing
.ft R             \" font
.sp 0.5v          \" half a line
.ti +3m
..
```

This sets the font to Roman and the point size and line spacing to whatever values are stored in the number registers **PS** and **VS**.

Why are there two backslashes? This is the eternal problem of how to quote a quote. When **troff** originally reads the macro definition, it peels off one backslash to see what's coming next. To ensure that another is left in the definition when the macro is *used,* we have to put in two backslashes in the definition. If only one backslash is used, point size and vertical spacing will be frozen at the time the macro is defined, not when it is used.

Protecting by an extra layer of backslashes is only needed for **\n**, *****, **\$** (which we haven't come to yet), and **** itself. Things like **\s**, **\f**, **\h**, **\v**, and so on do not need an extra backslash, since they are converted by **troff** to an internal code immediately upon being seen.

Arithmetic expressions can appear anywhere that a number is expected. As a trivial example,

```
.nr PS \\n(PS−2
```

decrements PS by 2. Expressions can use the arithmetic operators +, −, *, /, % (mod), the relational operators >, >=, <, <=, =, and != (not equal), and parentheses.

Although the arithmetic we have done so far has been straightforward, more complicated things are somewhat tricky. First, number registers hold only integers. **troff** arithmetic uses truncating integer division, just like Fortran. Second, in the absence of parentheses, evaluation is done left-to-right without any operator precedence (including relational operators). Thus

```
7*−4+3/13
```

becomes '−1'. Number registers can occur anywhere in an expression, and so can scale indicators like **p**, **i**, **m**, and so on (but no spaces). Although integer division causes truncation, each number and its scale indicator is converted to machine units (1/432 inch) before any arithmetic is done, so 1i/2u evaluates to 0.5i correctly.

The scale indicator **u** often has to appear when you wouldn't expect it — in particular, when arithmetic is being done in a context that implies horizontal or vertical dimensions. For example,

```
.ll 7/2i
```

would seem obvious enough — 3½ inches. Sorry. Remember that the default units for horizontal parameters like **.ll** are ems. That's really '7 ems / 2 inches', and when translated into machine units, it becomes zero. How about

```
.ll 7i/2
```

Sorry, still no good — the '2' is '2 ems', so '7i/2' is small, although not zero. You *must* use

```
.ll 7i/2u
```

So again, a safe rule is to attach a scale indicator to every number, even constants.

For arithmetic done within a **.nr** command, there is no implication of horizontal or vertical dimension, so the default units are 'units', and 7i/2 and 7i/2u mean the same thing. Thus

```
.nr ll 7i/2
.ll \\n(llu
```

does just what you want, so long as you don't forget the **u** on the **.ll** command.

11. Macros with arguments

The next step is to define macros that can change from one use to the next according to parameters supplied as arguments. To make this work, we need two things: first, when we define the macro, we have to indicate that some parts of it will be provided as arguments when the macro is called. Then when the macro is called we have to provide actual arguments to be plugged into the definition.

Let us illustrate by defining a macro **.SM** that will print its argument two points smaller than the surrounding text. That is, the macro call

```
.SM TROFF
```

will produce TROFF.

The definition of **.SM** is

```
.de SM
\s−2\\$1\s+2
..
```

Within a macro definition, the symbol **\\$n** refers to the **n**th argument that the macro was called with. Thus **\\$1** is the string to be placed in a smaller point size when **.SM** is called.

As a slightly more complicated version, the following definition of **.SM** permits optional second and third arguments that will be printed in the normal size:

```
.de SM
\\$3\s−2\\$1\s+2\\$2
..
```

Arguments not provided when the macro is called are treated as empty, so

```
.SM  TROFF  ),
```

produces TROFF), while

```
.SM  TROFF  ).  (
```

produces (TROFF). It is convenient to reverse the order of arguments because trailing punctuation is much more common than leading.

By the way, the number of arguments that a macro was called with is available in number register **.$**.

The following macro **.BD** is the one used to make the 'bold roman' we have been using for **troff** command names in text. It combines horizontal motions, width computations, and argument rearrangement.

```
.de BD
\&\\$3\f1\\$1\h´−\w´\\$1´u+1u´\\$1\fP\\$2
..
```

The **\h** and **\w** commands need no extra backslash, as we discussed above. The **\&** is there in case the argument begins with a period.

Two backslashes are needed with the **\\$n** commands, though, to protect one of them when the macro is being defined. Perhaps a second example will make this clearer. Consider a macro called **.SH** which produces section headings rather like those in this paper, with the sections numbered automatically, and the title in bold in a smaller size. The use is

```
.SH  "Section title ..."
```

(If the argument to a macro is to contain blanks, then it must be *surrounded* by double quotes, unlike a string, where only one leading quote is permitted.)

Here is the definition of the **.SH** macro:

```
.nr SH 0        \" initialize section number
.de SH
.sp 0.3i
.ft B
.nr SH \\n(SH+1        \" increment number
.ps \\n(PS−1   \" decrease PS
\\n(SH.  \\$1   \" number. title
.ps \\n(PS                \" restore PS
.sp 0.3i
.ft R
..
```

The section number is kept in number register SH, which is incremented each time just before it is used. (A number register may have the same name as a macro without conflict but a string may not.)

We used **\\n(SH** instead of **\n(SH** and **\\n(PS** instead of **\n(PS**. If we had used **\n(SH**, we would get the value of the register at the time the macro was *defined,* not at the time it was *used.* If that's what you want, fine, but not here. Similarly, by using **\\n(PS**, we get the point size at the time the macro is called.

As an example that does not involve numbers, recall our **.NP** macro which had a

```
.tl  ´left´center´right´
```

We could make these into parameters by using instead

```
.tl  ´\\*(LT´\\*(CT´\\*(RT´
```

so the title comes from three strings called LT, CT and RT. If these are empty, then the title will be a blank line. Normally CT would be set with something like

```
.ds  CT  - % -
```

to give just the page number between hyphens (as on the top of this page), but a user could supply private definitions for any of the strings.

12. Conditionals

Suppose we want the .SH macro to leave two extra inches of space just before section 1, but nowhere else. The cleanest way to do that is to test inside the .SH macro whether the section number is 1, and add some space if it is. The .if command provides the conditional test that we can add just before the heading line is output:

```
.if \\n(SH=1 .sp 2i        \" first section only
```

The condition after the .if can be any arithmetic or logical expression. If the condition is logically true, or arithmetically greater than zero, the rest of the line is treated as if it were text — here a command. If the condition is false, or zero or negative, the rest of the line is skipped.

It is possible to do more than one command if a condition is true. Suppose several operations are to be done before section 1. One possibility is to define a macro .S1 and invoke it if we are about to do section 1 (as determined by an .if).

```
.de S1
--- processing for section 1 ---
..
.de SH
...
.if \\n(SH=1 .S1
...
..
```

An alternate way is to use the extended form of the .if, like this:

```
.if \\n(SH=1 \{--- processing
for section 1 ----\}
```

The braces \{ and \} must occur in the positions shown or you will get unexpected extra lines in your output. **troff** also provides an 'if-else' construction, which we will not go into here.

A condition can be negated by preceding it with !; we get the same effect as above (but less clearly) by using

```
.if !\\n(SH>1 .S1
```

There are a handful of other conditions that can be tested with .if. For example, is the current page even or odd?

```
.if e .tl ´´even page title´´
.if o .tl ´´odd page title´´
```

gives facing pages different titles when used inside an appropriate new page macro.

Two other conditions are **t** and **n**, which tell you whether the formatter is **troff** or **nroff**.

```
.if t troff stuff ...
.if n nroff stuff ...
```

Finally, string comparisons may be made in an .if:

```
.if ´string1´string2´ stuff
```

does 'stuff' if *string1* is the same as *string2*. The character separating the strings can be anything reasonable that is not contained in either string. The strings themselves can reference strings with *, arguments with \$, and so on.

13. Environments

As we mentioned, there is a potential problem when going across a page boundary: parameters like size and font for a page title may well be different from those in effect in the text when the page boundary occurs. **troff** provides a very general way to deal with this and similar situations. There are three 'environments', each of which has independently settable versions of many of the parameters associated with processing, including size, font, line and title lengths, fill/nofill mode, tab stops, and even partially collected lines. Thus the titling problem may be readily solved by processing the main text in one environment and titles in a separate one with its own suitable parameters.

The command **.ev n** shifts to environment **n**; **n** must be 0, 1 or 2. The command **.ev** with no argument returns to the previous environment. Environment names are maintained in a stack, so calls for different environments may be nested and unwound consistently.

Suppose we say that the main text is processed in environment 0, which is where **troff** begins by default. Then we can modify the new page macro **.NP** to process titles in environment 1 like this:

```
.de NP
.ev 1         \" shift to new environment
.lt 6i        \" set parameters here
.ft R
.ps 10
... any other processing ...
.ev           \" return to previous environment
..
```

It is also possible to initialize the parameters for an environment outside the **.NP** macro, but the version shown keeps all the processing in one place and is thus easier to understand and change.

14. Diversions

There are numerous occasions in page layout when it is necessary to store some text for a period of time without actually printing it. Footnotes are the most obvious example: the text of the footnote usually appears in the input well before the place on the page where it is to be printed is reached. In fact, the place where it is output normally depends on how big it is, which implies that there must be a way to process the footnote at least enough to decide its size without printing it.

troff provides a mechanism called a diversion for doing this processing. Any part of the output may be diverted into a macro instead of being printed, and then at some convenient time the macro may be put back into the input.

The command **.di xy** begins a diversion — all subsequent output is collected into the macro **xy** until the command **.di** with no arguments is encountered. This terminates the diversion. The processed text is available at any time thereafter, simply by giving the command

```
.xy
```

The vertical size of the last finished diversion is contained in the built-in number register **dn**.

As a simple example, suppose we want to implement a 'keep-release' operation, so that text between the commands **.KS** and **.KE** will not be split across a page boundary (as for a figure or table). Clearly, when a **.KS** is encountered, we have to begin diverting the output so we can find out how big it is. Then when a **.KE** is seen, we decide whether the diverted text will fit on the current page, and print it either there if it fits, or at the top of the next page if it doesn't. So:

```
.de KS      \" start keep
.br         \" start fresh line
.ev 1       \" collect in new environment
.fi         \" make it filled text
.di XX      \" collect in XX
..

.de KE      \" end keep
.br         \" get last partial line
.di         \" end diversion
.if \\n(dn>=\\n(.t .bp   \" bp if doesn´t fit
.nf         \" bring it back in no-fill
.XX         \" text
.ev         \" return to normal environment
..
```

Recall that number register **nl** is the current position on the output page. Since output was being diverted, this remains at its value when the diversion started. **dn** is the amount of text in the diversion; **.t** (another built-in register) is the distance to the next trap, which we assume is at the bottom margin of the page. If the diversion is large enough to go past the trap, the **.if** is satisfied, and a **.bp** is issued. In either case, the diverted output is then brought back with **.XX**. It is essential to bring it back in no-fill mode so **troff** will do no further processing on it.

This is not the most general keep-release, nor is it robust in the face of all conceivable inputs, but it would require more space than we have here to write it in full generality. This section is not intended to teach everything about diversions, but to sketch out enough that you can read existing macro packages with some comprehension.

Acknowledgements

I am deeply indebted to J. F. Ossanna, the author of **troff**, for his repeated patient explanations of fine points, and for his continuing willingness to adapt **troff** to make other uses easier. I am also grateful to Jim Blinn, Ted Dolotta, Doug McIlroy, Mike Lesk and Joel Sturman for helpful comments on this paper.

References

[1] J. F. Ossanna, *NROFF/TROFF* User's Manual, Bell Laboratories Computing Science Technical Report 54, 1976.

[2] B. W. Kernighan, *A System for Typesetting Mathematics — User's Guide (Second Edition),* Bell Laboratories Computing Science Technical Report 17, 1977.

[3] M. E. Lesk, *TBL — A Program to Format Tables,* Bell Laboratories Computing Science Technical Report 49, 1976.

[4] M. E. Lesk, *Typing Documents on UNIX,* Bell Laboratories, 1978.

[5] J. R. Mashey and D. W. Smith, *PWB/MM — Programmer's Workbench Memorandum Macros,* Bell Laboratories internal memorandum.

Appendix A: Phototypesetter Character Set

These characters exist in roman, italic, and bold. To get the one on the left, type the four-character name on the right.

ff	\(ff	fi	\(fi	fl	\(fl	ffi	\(Fi	ffl	\(Fl
_	\(ru	—	\(em	¼	\(14	½	\(12	¾	\(34
©	\(co	°	\(de	†	\(dg	′	\(fm	¢	\(ct
®	\(rg	•	\(bu	□	\(sq	-	\(hy		

(In bold, \(sq is ■.)

The following are special-font characters:

+	\(pl	−	\(mi	×	\(mu	÷	\(di
=	\(eq	≡	\(==	≥	\(>=	≤	\(<=
≠	\(!=	±	\(+-	¬	\(no	⁄	\(sl
~	\(ap	≃	\(˜=	∝	\(pt	∇	\(gr
→	\(->	←	\(<-	↑	\(ua	↓	\(da
∫	\(is	∂	\(pd	∞	\(if	√	\(sr
⊂	\(sb	⊃	\(sp	∪	\(cu	∩	\(ca
⊆	\(ib	⊇	\(ip	∈	\(mo	∅	\(es
´	\(aa	`	\(ga	○	\(ci	🔔	\(bs
§	\(sc	‡	\(dd	☜	\(lh	☞	\(rh
⎡	\(lt	⎤	\(rt	⌈	\(lc	⌉	\(rc
⎣	\(lb	⎦	\(rb	⌊	\(lf	⌋	\(rf
⎨	\(lk	⎬	\(rk	│	\(bv	ς	\(ts
│	\(br	|	\(or	_	\(ul	‾	\(rn
∗	\(**						

These four characters also have two-character names. The ´ is the apostrophe on terminals; the ` is the other quote mark.

´	\´	`	\`	–	\–	_	_

These characters exist only on the special font, but they do not have four-character names:

" { } < > ˜ ˆ \ # @

For greek, precede the roman letter by \(* to get the corresponding greek; for example, \(*a is α.

a b g d e z y h i k l m n c o p r s t u f x q w
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ τ υ φ χ ψ ω

A B G D E Z Y H I K L M N C O P R S T U F X Q W
Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν Ξ Ο Π Ρ Σ Τ Υ Φ Χ Ψ Ω

C Reference Manual

Dennis M. Ritchie
Bell Telephone Laboratories
Murray Hill, New Jersey 07974

1. Introduction

C is a computer language based on the earlier language B [1]. The languages and their compilers differ in two major ways: C introduces the notion of types, and defines appropriate extra syntax and semantics; also, C on the PDP-11 is a true compiler, producing machine code where B produced interpretive code.

Most of the software for the UNIX time-sharing system [2] is written in C, as is the operating system itself. C is also available on the HIS 6070 computer at Murray Hill and and on the IBM System/370 at Holmdel [3]. This paper is a manual only for the C language itself as implemented on the PDP-11. However, hints are given occasionally in the text of implementation-dependent features.

The UNIX Programmer's Manual [4] describes the library routines available to C programs under UNIX, and also the procedures for compiling programs under that system. ''The GCOS C Library'' by Lesk and Barres [5] describes routines available under that system as well as compilation procedures. Many of these routines, particularly the ones having to do with I/O, are also provided under UNIX. Finally, ''Programming in C− A Tutorial,'' by B. W. Kernighan [6], is as useful as promised by its title and the author's previous introductions to allegedly impenetrable subjects.

2. Lexical conventions

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1 Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`.

2.2 Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore ''_'' counts as alphabetic. Upper and lower case letters are considered different. No more than the first eight characters are significant, and only the first seven for external identifiers.

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
int             break
char            continue
float           if
double          else
struct          for
auto            do
extern          while
register        switch
static          case
goto            default
return          entry
sizeof
```

The `entry` keyword is not currently implemented by any compiler but is reserved for future use.

2.3 Constants

There are several kinds of constants, as follows:

2.3.1 Integer constants

An integer constant is a sequence of digits. An integer is taken to be octal if it begins with `0`, decimal otherwise. The digits `8` and `9` have octal value 10 and 11 respectively.

2.3.2 Character constants

A character constant is 1 or 2 characters enclosed in single quotes ‘‘ ´ ’’. Within a character constant a single quote must be preceded by a back-slash ‘‘\’’. Certain non-graphic characters, and ‘‘\’’ itself, may be escaped according to the following table:

BS	\b
NL	\n
CR	\r
HT	\t
ddd	*ddd*
\	\\

The escape ‘‘*ddd*’’ consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is ‘‘\0’’ (not followed by a digit) which indicates a null character.

Character constants behave exactly like integers (not, in particular, like objects of character type). In conformity with the addressing structure of the PDP-11, a character constant of length 1 has the code for the given character in the low-order byte and 0 in the high-order byte; a character constant of length 2 has the code for the first character in the low byte and that for the second character in the high-order byte. Character constants with more than one character are inherently machine-dependent and should be avoided.

2.3.3 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an `e`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the `e` and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

2.4 Strings

A string is a sequence of characters surrounded by double quotes ‘‘ " ’’. A string has the type array-of-characters (see below) and refers to an area of storage initialized with the given characters. The compiler places a null byte (\0) at the end of each string so that programs which scan the string can find its end. In a string, the character ‘‘ " ’’ must be preceded by a ‘‘\’’ ; in addition, the same escapes as described for character constants may be used.

3. Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in `gothic`. Alternatives are listed on separate lines. An optional terminal or non-terminal symbol is indicated by the subscript ''opt,'' so that

$$\{\ expression_{opt}\ \}$$

would indicate an optional expression in braces.

4. What's in a Name?

C bases the interpretation of an identifier upon two attributes of the identifier: its *storage class* and its *type.* The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a function, and are discarded on return; static variables are local to a function, but retain their values independently of invocations of the function; external variables are independent of any function. Register variables are stored in the fast registers of the machine; like automatic variables they are local to each function and disappear on return.

C supports four fundamental types of objects: characters, integers, single-, and double-precision floating-point numbers.

> Characters (declared, and hereinafter called, `char`) are chosen from the ASCII set; they occupy the rightmost seven bits of an 8-bit byte. It is also possible to interpret `chars` as signed, 2's complement 8-bit numbers.
>
> Integers (`int`) are represented in 16-bit 2's complement notation.
>
> Single precision floating point (`float`) quantities have magnitude in the range approximately $10^{\pm 38}$ or 0; their precision is 24 bits or about seven decimal digits.
>
> Double-precision floating-point (`double`) quantities have the same range as `floats` and a precision of 56 bits or about 17 decimal digits.

Besides the four fundamental types there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

> *arrays* of objects of most types;
>
> *functions* which return objects of a given type;
>
> *pointers* to objects of a given type;
>
> *structures* containing objects of various types.

In general these methods of constructing objects can be applied recursively.

5. Objects and lvalues

An object is a manipulatable region of storage; an lvalue is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if E is an expression of pointer type, then ∗E is an lvalue expression referring to the object to which E points. The name ''lvalue'' comes from the assignment expression ''E1 = E2'' in which the left operand E1 must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

6. Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions.

6.1 Characters and integers

A `char` object may be used anywhere an `int` may be. In all cases the `char` is converted to an `int` by propagating its sign through the upper 8 bits of the resultant integer. This is consistent with the two's complement representation used for both characters and integers. (However, the sign-propagation feature disappears in other implementations.)

6.2 Float and double

All floating arithmetic in C is carried out in double-precision; whenever a `float` appears in an expression it is lengthened to `double` by zero-padding its fraction. When a `double` must be converted to `float`, for example by an assignment, the `double` is rounded before truncation to `float` length.

6.3 Float and double; integer and character

All `ints` and `chars` may be converted without loss of significance to `float` or `double`. Conversion of `float` or `double` to `int` or `char` takes place with truncation towards 0. Erroneous results can be expected if the magnitude of the result exceeds 32,767 (for `int`) or 127 (for `char`).

6.4 Pointers and integers

Integers and pointers may be added and compared; in such a case the `int` is converted as specified in the discussion of the addition operator.

Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Thus the expressions referred to as the operands of + (§7.4) are those expressions defined in §§7.1_7.3. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized in an appendix.

Otherwise the order of evaluation of expressions is undefined. In particular the compiler considers itself free to compute subexpressions in the order it believes most efficient, even if the subexpressions involve side effects.

7.1 Primary expressions

Primary expressions involving `.`, −>, subscripting, and function calls group left to right.

7.1.1 *identifier*

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration. However, if the type of the identifier is ‘‘array of ...’’, then the value of the identifier-expression is a pointer to the first object in the array, and the type of the expression is ‘‘pointer to ...’’. Moreover, an array identifier is not an lvalue expression.

Likewise, an identifier which is declared ‘‘function returning ...’’, when used except in the function-name position of a call, is converted to ‘‘pointer to function returning ...’’.

7.1.2 *constant*

A decimal, octal, character, or floating constant is a primary expression. Its type is `int` in the first three cases, `double` in the last.

7.1.3 *string*

A string is a primary expression. Its type is originally ‘‘array of `char`’’; but following the same rule as in §7.1.1 for identifiers, this is modified to ‘‘pointer to `char`’’ and the result is a pointer to the first character in the string.

7.1.4 (*expression*)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

7.1.5 *primary-expression* [*expression*]

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type ''pointer to ...'', the subscript expression is `int`, and the type of the result is ''...''. The expression ''E1[E2]'' is identical (by definition) to ''∗((E1)+(E2))''. All the clues needed to understand this notation are contained in this section together with the discussions in §§ 7.1.1, 7.2.1, and 7.4.1 on identifiers, ∗, and + respectively; §14.3 below summarizes the implications.

7.1.6 *primary-expression* (*expression-list*$_{opt}$)

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type ''function returning ...'', and the result of the function call is of type ''...''. As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type `float` are converted to `double` before the call; any of type `char` are converted to `int`.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot possibly affect the values of the actual parameters. On the other hand, it is perfectly possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points.

Recursive calls to any function are permissible.

7.1.7 *primary-lvalue* **.** *member-of-structure*

An lvalue expression followed by a dot followed by the name of a member of a structure is a primary expression. The object referred to by the lvalue is assumed to have the same form as the structure containing the structure member. The result of the expression is an lvalue appropriately offset from the origin of the given lvalue whose type is that of the named structure member. The given lvalue is not required to have any particular type.

Structures are discussed in §8.5.

7.1.8 *primary-expression* −> *member-of-structure*

The primary-expression is assumed to be a pointer which points to an object of the same form as the structure of which the member-of-structure is a part. The result is an lvalue appropriately offset from the origin of the pointed-to structure whose type is that of the named structure member. The type of the primary-expression need not in fact be pointer; it is sufficient that it be a pointer, character, or integer.

Except for the relaxation of the requirement that E1 be of pointer type, the expression ''E1−>MOS'' is exactly equivalent to ''(∗E1).MOS''.

7.2 Unary operators

Expressions with unary operators group right-to-left.

7.2.1 ∗ *expression*

The unary ∗ operator means *indirection:* the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is ''pointer to ...'', the type of the result is ''...''.

7.2.2 & *lvalue-expression*

The result of the unary & operator is a pointer to the object referred to by the lvalue-expression. If the type of the lvalue-expression is ''...'', the type of the result is ''pointer to ...''.

7.2.3 − *expression*

The result is the negative of the expression, and has the same type. The type of the expression must be `char`, `int`, `float`, or `double`.

7.2.4 ! *expression*

The result of the logical negation operator ! is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero. The type of the result is `int`. This operator is applicable only to `int`s or `char`s.

7.2.5 ~ *expression*

The ~ operator yields the one's complement of its operand. The type of the expression must be `int` or `char`, and the result is `int`.

7.2.6 ++ *lvalue-expression*

The object referred to by the lvalue expression is incremented. The value is the new value of the lvalue expression and the type is the type of the lvalue. If the expression is `int` or `char`, it is incremented by 1; if it is a pointer to an object, it is incremented by the length of the object. ++ is applicable only to these types. (Not, for example, to `float` or `double`.)

7.2.7 –– *lvalue-expression*

The object referred to by the lvalue expression is decremented analogously to the ++ operator.

7.2.8 *lvalue-expression* ++

The result is the value of the object referred to by the lvalue expression. After the result is noted, the object referred to by the lvalue is incremented in the same manner as for the prefix ++ operator: by 1 for an `int` or `char`, by the length of the pointed-to object for a pointer. The type of the result is the same as the type of the lvalue-expression.

7.2.9 *lvalue-expression* ––

The result of the expression is the value of the object referred to by the the lvalue expression. After the result is noted, the object referred to by the lvalue expression is decremented in a way analogous to the postfix ++ operator.

7.2.10 `sizeof` *expression*

The `sizeof` operator yields the size, in bytes, of its operand. When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an integer constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

7.3 Multiplicative operators

The multiplicative operators *, `/`, and `%` group left-to-right.

7.3.1 *expression* * *expression*

The binary * operator indicates multiplication. If both operands are `int` or `char`, the result is `int`; if one is `int` or `char` and one `float` or `double`, the former is converted to `double`, and the result is `double`; if both are `float` or `double`, the result is `double`. No other combinations are allowed.

7.3.2 *expression* / *expression*

The binary / operator indicates division. The same type considerations as for multiplication apply.

7.3.3 *expression* % *expression*

The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be `int` or `char`, and the result is `int`. In the current implementation, the remainder has the same sign as the dividend.

7.4 Additive operators

The additive operators + and – group left-to-right.

7.4.1 *expression* + *expression*

The result is the sum of the expressions. If both operands are `int` or `char`, the result is `int`. If both are `float` or `double`, the result is `double`. If one is `char` or `int` and one is `float` or `double`, the former is converted to `double` and the result is `double`. If an `int` or `char` is added to a pointer, the former is converted by multiplying it by the length of the object to which the pointer points and the result is a pointer of the same type as the original pointer. Thus if P is a pointer to an object, the expression ‘‘P+1’’ is a pointer to another object of the same type as the first and immediately following it in storage.

No other type combinations are allowed.

7.4.2 *expression* − *expression*

The result is the difference of the operands. If both operands are `int`, `char`, `float`, or `double`, the same type considerations as for + apply. If an `int` or `char` is subtracted from a pointer, the former is converted in the same way as explained under + above.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an `int` representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

7.5 Shift operators

The shift operators << and >> group left-to-right.

7.5.1 *expression* << *expression*
7.5.2 *expression* >> *expression*

Both operands must be `int` or `char`, and the result is `int`. The second operand should be non-negative. The value of ‘‘E1<<E2’’ is E1 (interpreted as a bit pattern 16 bits long) left-shifted E2 bits; vacated bits are 0-filled. The value of ‘‘E1>>E2’’ is E1 (interpreted as a two’s complement, 16-bit quantity) arithmetically right-shifted E2 bit positions. Vacated bits are filled by a copy of the sign bit of E1. [Note: the use of arithmetic rather than logical shift does not survive transportation between machines.]

7.6 Relational operators

The relational operators group left-to-right, but this fact is not very useful; ‘‘a<b<c’’ does not mean what it seems to.

7.6.1 *expression* < *expression*
7.6.2 *expression* > *expression*
7.6.3 *expression* <= *expression*
7.6.4 *expression* >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. Operand conversion is exactly the same as for the + operator except that pointers of any kind may be compared; the result in this case depends on the relative locations in storage of the pointed-to objects. It does not seem to be very meaningful to compare pointers with integers other than 0.

7.7 Equality operators

7.7.1 *expression* == *expression*
7.7.2 *expression* != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus ‘‘a<b == c<d’’ is 1 whenever a<b and c<d have the same truth-value).

7.8 *expression* & *expression*

The & operator groups left-to-right. Both operands must be `int` or `char`; the result is an `int` which is the bit-wise logical `and` function of the operands.

7.9 *expression* ^ *expression*

The ^ operator groups left-to-right. The operands must be `int` or `char`; the result is an `int` which is the bit-wise exclusive `or` function of its operands.

7.10 *expression* | *expression*

The | operator groups left-to-right. The operands must be `int` or `char`; the result is an `int` which is the bit-wise inclusive `or` of its operands.

7.11 *expression* `&&` *expression*

The `&&` operator returns 1 if both its operands are non-zero, 0 otherwise. Unlike `&`, `&&` guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer.

7.12 *expression* || *expression*

The || operator returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike `|`, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer.

7.13 *expression* `?` *expression* **:** *expression*

Conditional expressions group left-to-right. The first expression is evaluated and if it is non-zero, the result is the value of the second expression, otherwise that of third expression. If the types of the second and third operand are the same, the result has their common type; otherwise the same conversion rules as for + apply. Only one of the second and third expressions is evaluated.

7.14 Assignment operators

There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.

7.14.1 *lvalue* = *expression*

The value of the expression replaces that of the object referred to by the lvalue. The operands need not have the same type, but both must be `int`, `char`, `float`, `double`, or pointer. If neither operand is a pointer, the assignment takes place as expected, possibly preceded by conversion of the expression on the right.

When both operands are `int` or pointers of any kind, no conversion ever takes place; the value of the expression is simply stored into the object referred to by the lvalue. Thus it is possible to generate pointers which will cause addressing exceptions when used.

7.14.2 *lvalue* =+ *expression*
7.14.3 *lvalue* =− *expression*
7.14.4 *lvalue* =∗ *expression*
7.14.5 *lvalue* =/ *expression*
7.14.6 *lvalue* =% *expression*
7.14.7 *lvalue* =>> *expression*
7.14.8 *lvalue* =<< *expression*
7.14.9 *lvalue* =& *expression*
7.14.10 *lvalue* =^ *expression*
7.14.11 *lvalue* =| *expression*

The behavior of an expression of the form ‘‘E1 =op E2’’ may be inferred by taking it as equivalent to ‘‘E1 = E1 op E2’’; however, E1 is evaluated only once. Moreover, expressions like ‘‘i =+ p’’ in which a pointer is added to an integer, are forbidden.

7.15 *expression* `,` *expression*

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. It should be avoided in situations where comma is given a special meaning, for example in actual arguments to function calls (§7.1.6) and lists of initializers (§10.2).

8. Declarations

Declarations are used within function definitions to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

> *declaration:*
> > *decl-specifiers declarator-list*$_{opt}$ `;`

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of at most one type-specifier and at most one storage class specifier.

> *decl-specifiers:*
> > *type-specifier*
> > *sc-specifier*
> > *type-specifier sc-specifier*
> > *sc-specifier type-specifier*

8.1 Storage class specifiers

The sc-specifiers are:

> *sc-specifier:*
> > `auto`
> > `static`
> > `extern`
> > `register`

The `auto`, `static`, and `register` declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the `extern` case there must be an external definition (see below) for the given identifiers somewhere outside the function in which they are declared.

There are some severe restrictions on `register` identifiers: there can be at most 3 register identifiers in any function, and the type of a register identifier can only be `int`, `char`, or pointer (not `float`, `double`, structure, function, or array). Also the address-of operator `&` cannot be applied to such identifiers. Except for these restrictions (in return for which one is rewarded with faster, smaller code), register identifiers behave as if they were automatic. In fact implementations of C are free to treat `register` as synonymous with `auto`.

If the sc-specifier is missing from a declaration, it is generally taken to be `auto`.

8.2 Type specifiers

The type-specifiers are

> *type-specifier:*
> > `int`
> > `char`
> > `float`
> > `double`
> > `struct` *{ type-decl-list }*
> > `struct` *identifier { type-decl-list }*
> > `struct` *identifier*

The `struct` specifier is discussed in §8.5. If the type-specifier is missing from a declaration, it is generally taken to be `int`.

8.3 Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators.

> *declarator-list:*
>
> *declarator*
>
> *declarator* **,** *declarator-list*

The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

> *declarator:*
>
> *identifier*
>
> ∗ *declarator*
>
> *declarator* ( )
>
> *declarator* [*constant-expression*$_{opt}$]
>
> (*declarator*)

The grouping in this definition is the same as in expressions.

8.4 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class. Each declarator contains exactly one identifier; it is this identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

If a declarator has the form

> ∗ D

for D a declarator, then the contained identifier has the type ‘‘pointer to . . .’’, where ‘‘. . .’’ is the type which the identifier would have had if the declarator had been simply D.

If a declarator has the form

> D( )

then the contained identifier has the type ‘‘function returning ...’’, where ‘‘. . .’’ is the type which the identifier would have had if the declarator had been simply D.

A declarator may have the form

> D[constant-expression]

or

> D[ ]

In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is `int.` in the second the constant 1 is used. (Constant expressions are defined precisely in §15.) Such a declarator makes the contained identifier have type ‘‘array.’’ If the unadorned declarator D would specify a non-array of type ‘‘. . .’’, then the declarator ‘‘D[i]’’ yields a 1-dimensional array with rank *i* of objects of type ‘‘. . .’’. If the unadorned declarator D would specify an n-dimensional array with rank $i_1 \times i_2 \times \ldots \times i_n$, then the declarator ‘‘D[i_{n+1}]’’ yields an $(n+1)$-dimensional array with rank $i_1 \times i_2 \times \ldots \times i_n \times i_{n+1}$.

An array may be constructed from one of the basic types, from a pointer, from a structure, or from another array (to generate a multi-dimensional array).

Finally, parentheses in declarators do not alter the type of the contained identifier except insofar as they alter the binding of the components of the declarator.

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays, structures or functions, although they may return pointers to such things; there are no arrays of functions, although there may be arrays of pointers to functions. Likewise a structure may not contain a function, but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares an integer *i*, a pointer *ip* to an integer, a function *f* returning an integer, a function *fip* returning a pointer to an integer, and a pointer *pfi* to a function which returns an integer. Also

```
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank 3×5×7. In complete detail, *x3d* is an array of three items: each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions ''x3d'', ''x3d[i]'', ''x3d[i][j]'', ''x3d[i][j][k]'' may reasonably appear in an expression. The first three have type ''array'', the last has type `int`.

8.5 Structure declarations

Recall that one of the forms for a structure specifier is

`struct` *{ type-decl-list }*

The *type-decl-list* is a sequence of type declarations for the members of the structure:

type-decl-list:
type-declaration
type-declaration type-decl-list

A type declaration is just a declaration which does not mention a storage class (the storage class ''member of structure'' here being understood by context).

type-declaration:
type-specifier declarator-list `;`

Within the structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each component of a structure begins on an addressing boundary appropriate to its type. On the PDP-11 the only requirement is that non-characters begin on a word boundary; therefore, there may be 1-byte, unnamed holes in a structure, and all structures have an even length in bytes.

Another form of structure specifier is

`struct` *identifier { type-decl-list }*

This form is the same as the one just discussed, except that the identifier is remembered as the *structure tag* of the structure specified by the list. A subsequent declaration may then be given using the structure tag but without the list, as in the third form of structure specifier:

`struct` *identifier*

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is however absurd to declare a structure which contains an instance of itself, as distinct from a pointer to an instance of itself.

A simple example of a structure declaration, taken from §16.2 where its use is illustrated more fully, is

```
struct tnode {
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the following declaration makes sense:

```
struct tnode s, *sp;
```

which declares *s* to be a structure of the given sort and *sp* to be a pointer to a structure of the given sort.

The names of structure members and structure tags may be the same as ordinary variables, since a distinction can be made by context. However, names of tags and members must be distinct. The same member name can appear in different structures only if the two members are of the same type and if their origin with respect to their structure is the same; thus separate structures can share a common initial segment.

9. Statements

Except as indicated, statements are executed in sequence.

9.1 Expression statement

Most statements are expression statements, which have the form

> *expression* `;`

Usually expression statements are assignments or function calls.

9.2 Compound statement

So that several statements can be used where one is expected, the compound statement is provided:

> *compound-statement:*
>   *{ statement-list }*
>
> *statement-list:*
>   *statement*
>   *statement statement-list*

9.3 Conditional statement

The two forms of the conditional statement are

> `if (` *expression* `)` *statement*
> `if (` *expression* `)` *statement* `else` *statement*

In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the ‘‘else’’ ambiguity is resolved by connecting an `else` with the last encountered elseless `if`.

9.4 While statement

The `while` statement has the form

> `while (` *expression* `)` *statement*

The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

9.5 Do statement

The `do` statement has the form

> `do` *statement* `while (` *expression* `) ;`

The substatement is executed repeatedly until the value of the expression becomes zero. The test takes place after each execution of the statement.

9.6 For statement

The `for` statement has the form

`for (` *expression-1*$_{opt}$ `;` *expression-2*$_{opt}$ `;` *expression-3*$_{opt}$ `)` *statement*

This statement is equivalent to

```
expression-1;
while (expression-2) {
        statement
        expression-3;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression typically specifies an incrementation which is performed after each iteration.

Any or all of the expressions may be dropped. A missing *expression-2* makes the implied `while` clause equivalent to ''while(1)''; other missing expressions are simply dropped from the expansion above.

9.7 Switch statement

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

`switch (` *expression* `)` *statement*

The expression must be `int` or `char`. The statement is typically compound. Each statement within the statement may be labelled with case prefixes as follows:

`case` *constant-expression* `:`

where the constant expression must be `int` or `char`. No two of the case constants in a switch may have the same value. Constant expressions are precisely defined in §15.

There may also be at most one statement prefix of the form

```
default :
```

When the `switch` statement is executed, its expression is evaluated and compared with each case constant in an undefined order. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a `default` prefix, control passes to the prefixed statement. In the absence of a `default` prefix none of the statements in the switch is executed.

Case or default prefixes in themselves do not alter the flow of control.

9.8 Break statement

The statement

```
break ;
```

causes termination of the smallest enclosing `while`, `do`, `for`, or `switch` statement; control passes to the statement following the terminated statement.

9.9 Continue statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing `while`, `do`, or `for` statement; that is to the end of the loop. More precisely, in each of the statements

```
while ( . . . ) {          do {                      for ( . . . ) {
   . . .                      . . .                     . . .
contin: ;                  contin: ;                 contin: ;
}                          } while ( . . . ) ;       }
```

a `continue` is equivalent to ‘‘goto contin’’.

9.10 Return statement

A function returns to its caller by means of the `return` statement, which has one of the forms

`return ;`
`return (` *expression* `) ;`

In the first case no value is returned. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

9.11 Goto statement

Control may be transferred unconditionally by means of the statement

`goto` *expression* `;`

The expression should be a label (§§9.12, 14.4) or an expression of type ‘‘pointer to `int`’’ which evaluates to a label. It is illegal to transfer to a label not located in the current function unless some extra-language provision has been made to adjust the stack correctly.

9.12 Labelled statement

Any statement may be preceded by label prefixes of the form

identifier `:`

which serve to declare the identifier as a label. More details on the semantics of labels are given in §14.4 below.

9.13 Null statement

The null statement has the form

`;`

A null statement is useful to carry a label just before the ‘‘}’’ of a compound statement or to supply a null body to a looping statement such as `while`.

10. External definitions

A C program consists of a sequence of external definitions. External definitions may be given for functions, for simple variables, and for arrays. They are used both to declare and to reserve storage for objects. An external definition declares an identifier to have storage class `extern` and a specified type. The type-specifier (§8.2) may be empty, in which case the type is taken to be `int`.

10.1 External function definitions

Function definitions have the form

function-definition:
  type-specifier$_{opt}$ *function-declarator function-body*

A function declarator is similar to a declarator for a ‘‘function returning ...’’ except that it lists the formal parameters of the function being defined.

function-declarator:
  declarator `(` *parameter-list*$_{opt}$ `)`

parameter-list:

identifier
identifier , *parameter-list*

The function-body has the form

function-body:
type-decl-list function-statement

The purpose of the type-decl-list is to give the types of the formal parameters. No other identifiers should be declared in this list, and formal parameters should be declared only here.

The function-statement is just a compound statement which may have declarations at the start.

function-statement:
{ declaration-list$_{opt}$ statement-list }

A simple example of a complete function definition is

```
int max(a, b, c)
int a, b, c;
{
        int m;
        m = (a>b)? a:b;
        return(m>c? m:c);
}
```

Here ''int'' is the type-specifier; ''max(a, b, c)'' is the function-declarator; ''int a, b, c;'' is the type-decl-list for the formal parameters; ''{ ... }'' is the function-statement.

C converts all `float` actual parameters to `double`, so formal parameters declared `float` have their declaration adjusted to read `double`. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared ''array of ...'' are adjusted to read ''pointer to ...''. Finally, because neither structures nor functions can be passed to a function, it is useless to declare a formal parameter to be a structure or function (pointers to structures or functions are of course permitted).

A free `return` statement is supplied at the end of each function definition, so running off the end causes control, but no value, to be returned to the caller.

10.2 External data definitions

An external data definition has the form

data-definition:
`extern`$_{opt}$ *type-specifier$_{opt}$ init-declarator-list$_{opt}$* ;

The optional `extern` specifier is discussed in § 11.2. If given, the init-declarator-list is a comma-separated list of declarators each of which may be followed by an initializer for the declarator.

init-declarator-list:
init-declarator
init-declarator , *init-declarator-list*

init-declarator:
declarator initializer$_{opt}$

Each initializer represents the initial value for the corresponding object being defined (and declared).

initializer:
constant
{ constant-expression-list }

constant-expression-list:
  constant-expression
  constant-expression **,** *constant-expression-list*

Thus an initializer consists of a constant-valued expression, or comma-separated list of expressions, inside braces. The braces may be dropped when the expression is just a plain constant. The exact meaning of a constant expression is discussed in §15. The expression list is used to initialize arrays; see below.

The type of the identifier being defined should be compatible with the type of the initializer: a `double` constant may initialize a `float` or `double` identifier; a non-floating-point expression may initialize an `int`, `char`, or pointer.

An initializer for an array may contain a comma-separated list of compile-time expressions. The length of the array is taken to be the maximum of the number of expressions in the list and the square-bracketed constant in the array's declarator. This constant may be missing, in which case 1 is used. The expressions initialize successive members of the array starting at the origin (subscript 0) of the array. The acceptable expressions for an array of type ''array of ...'' are the same as those for type ''...''. As a special case, a single string may be given as the initializer for an array of `char`s; in this case, the characters in the string are taken as the initializing values.

Structures can be initialized, but this operation is incompletely implemented and machine-dependent. Basically the structure is regarded as a sequence of words and the initializers are placed into those words. Structure initialization, using a comma-separated list in braces, is safe if all the members of the structure are integers or pointers but is otherwise ill-advised.

The initial value of any externally-defined object not explicitly initialized is guaranteed to be 0.

11. Scope rules

A complete C program need not all be compiled at the same time: the source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scope to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing ''undefined identifier'' diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

11.1 Lexical scope

C is not a block-structured language; this may fairly be considered a defect. The lexical scope of names declared in external definitions extends from their definition through the end of the file in which they appear. The lexical scope of names declared at the head of functions (either as formal parameters or in the declarations heading the statements constituting the function itself) is the body of the function.

It is an error to redeclare identifiers already declared in the current context, unless the new declaration specifies the same type and storage class as already possessed by the identifiers.

11.2 Scope of externals

If a function declares an identifier to be `extern`, then somewhere among the files or libraries constituting the complete program there must be an external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and extent specified in the definition are compatible with those specified by each function which references the data.

In PDP-11 C, it is explicitly permitted for (compatible) external definitions of the same identifier to be present in several of the separately-compiled pieces of a complete program, or even twice within the same program file, with the important limitation that the identifier may be initialized in at most one of the definitions. In other operating systems, however, the compiler must know in just which file the storage for the identifier is allocated, and in which file the identifier is merely being referred to. In the implementations of C for such systems, the appearance of the `extern` keyword before an external definition indicates that storage for the identifiers being declared will be allocated in another file. Thus in a multi-file program, an external data definition without the `extern` specifier must appear in exactly one of the files. Any other files which wish to give an external definition for the identifier must include the `extern` in the definition. The identifier can be initialized only in the file where storage is allocated.

In PDP-11 C none of this nonsense is necessary and the `extern` specifier is ignored in external definitions.

12. Compiler control lines

When a line of a C program begins with the character `#`, it is interpreted not by the compiler itself, but by a preprocessor which is capable of replacing instances of given identifiers with arbitrary token-strings and of inserting named files into the source program. In order to cause this preprocessor to be invoked, it is necessary that the very first line of the program begin with `#`. Since null lines are ignored by the preprocessor, this line need contain no other information.

12.1 Token replacement

A compiler-control line of the form

`# define` *identifier token-string*

(note: no trailing semicolon) causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens (except within compiler control lines). The replacement token-string has comments removed from it, and it is surrounded with blanks. No rescanning of the replacement string is attempted. This facility is most valuable for definition of ‘‘manifest constants’’, as in

```
# define tabsize 100
...
int table[tabsize];
```

12.2 File inclusion

Large C programs often contain many external data definitions. Since the lexical scope of external definitions extends to the end of the program file, it is good practice to put all the external definitions for data at the start of the program file, so that the functions defined within the file need not repeat tedious and error-prone declarations for each external identifier they use. It is also useful to put a heavily used structure definition at the start and use its structure tag to declare the `auto` pointers to the structure used within functions. To further exploit this technique when a large C program consists of several files, a compiler control line of the form

`# include "`*filename*`"`

results in the replacement of that line by the entire contents of the file *filename*.

13. Implicit declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. Sometimes the storage class is supplied by the context: in external definitions, and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be `int`; if a type but no storage class is indicated, the identifier is assumed to be `auto`. An exception to the latter rule is made for functions, since `auto` functions are meaningless (C being incapable of compiling code into the stack). If the type of an identifier is ‘‘function returning ...’’, it is implicitly declared to be `extern`.

In an expression, an identifier followed by `(` and not currently declared is contextually declared to be ‘‘function returning `int`’’.

Undefined identifiers not followed by `(` are assumed to be labels which will be defined later in the function. (Since a label is not an lvalue, this accounts for the ‘‘Lvalue required’’ error message sometimes noticed when an undeclared identifier is used.) Naturally, appearance of an identifier as a label declares it as such.

For some purposes it is best to consider formal parameters as belonging to their own storage class. In practice, C treats parameters as if they were automatic (except that, as mentioned above, formal parameter arrays and `float`s are treated specially).

14. Types revisited

This section summarizes the operations which can be performed on objects of certain types.

14.1 Structures

There are only two things that can be done with a structure: pick out one of its members (by means of the `.` or `->` operators); or take its address (by unary `&`). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message. In the future, it is expected that these operations, but not necessarily others, will be allowed.

14.2 Functions

There are only two things that can be done with a function: call it, or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f( );
...
g(f);
```

Then the definition of *g* might read

```
g(funcp)
int (*funcp)( );
{
        ...
        (*funcp)( );
        ...
}
```

Notice that *f* was declared explicitly in the calling routine since its first appearance was not followed by `(`.

14.3 Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator `[ ]` is interpreted in such a way that ''E1[E2]'' is identical to ''*((E1)+(E2))''. Because of the conversion rules which apply to +, if E1 is an array and E2 an integer, then E1[E2] refers to the E2-th member of E1. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multi-dimensional arrays. If E is an n-dimensional array of rank $i \times j \times \ldots \times k$, then E appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \ldots \times k$. If the $*$ operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here *x* is a 3×5 array of integers. When *x* appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression ''x[i]'', which is equivalent to ''*(x+i)'', *x* is first converted to a pointer as described; then *i* is converted to the type of *x*, which involves multiplying *i* by the length the object to which the pointer points, namely 5 integer objects. The results are added and indirection applied to yield an array (of 5 integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer.

It follows from all this that arrays in C are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations.

14.4 Labels

Labels do not have a type of their own; they are treated as having type ''array of `int`''. Label variables should be declared ''pointer to `int`''; before execution of a `goto` referring to the variable, a label (or an expression deriving from a label) should be assigned to the variable.

Label variables are a bad idea in general; the `switch` statement makes them almost always unnecessary.

15. Constant expressions

In several places C requires expressions which evaluate to a constant: after `case`, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, and `sizeof` expressions, possibly connected by the binary operators

```
+ − * / % & | ^ << >>
```

or by the unary operators

```
− ~
```

Parentheses can be used for grouping, but not for function calls.

A bit more latitude is permitted for initializers; besides constant expressions as discussed above, one can also apply the unary `&` operator to external scalars, and to external arrays subscripted with a constant expression. The unary `&` can also be applied implicitly by appearance of unsubscripted external arrays. The rule here is that initializers must evaluate either to a constant or to the address of an external identifier plus or minus a constant.

16. Examples.

These examples are intended to illustrate some typical C constructions as well as a serviceable style of writing C programs.

16.1 Inner product

This function returns the inner product of its array arguments.

```
double inner(v1, v2, n)
double v1[], v2[];
{
	double sum;
	int i;
	sum = 0.0;
	for (i=0; i<n; i++)
		sum =+ v1[i] * v2[i];
	return(sum);
}
```

The following version is somewhat more efficient, but perhaps a little less clear. It uses the facts that parameter arrays are really pointers, and that all parameters are passed by value.

```
double inner(v1, v2, n)
double *v1, *v2;
{
	double sum;
	sum = 0.0;
	while(n--)
		sum =+ *v1++ * *v2++;
	return(sum);
}
```

The declarations for the parameters are really exactly the same as in the last example. In the first case array declarations ‘‘[ ]’’ were given to emphasize that the parameters would be referred to as arrays; in the second, pointer declarations were given because the indirection operator and ++ were used.

16.2 Tree and character processing

Here is a complete C program (courtesy of R. Haight) which reads a document and produces an alphabetized list of words found therein together with the number of occurrences of each word. The method keeps a binary tree of words such that the left descendant tree for each word has all the words lexicographically smaller than the given word, and the right descendant has all the larger words. Both the insertion and the printing routine are recursive.

The program calls the library routines *getchar* to pick up characters and *exit* to terminate execution. *Printf* is called to print the results according to a format string. A version of *printf* is given below (§16.3).

Because all the external definitions for data are given at the top, no `extern` declarations are necessary within the functions. To stay within the rules, a type declaration is given for each non-integer function when the function is used before it is defined. However, since all such functions return pointers which are simply assigned to other pointers, no actual harm would result from leaving out the declarations; the supposedly `int` function values would be assigned without error or complaint.

```
# define nwords 100                    /* number of different words */
# define wsize 20                      /* max chars per word */
struct tnode {                         /* the basic structure */
      char tword[wsize];
      int count;
      struct tnode *left;
      struct tnode *right;
};

struct tnode space[nwords];            /* the words themselves */
int nnodes nwords;                     /* number of remaining slots */
struct tnode *spacep space;            /* next available slot */
struct tnode *freep;                   /* free list */
/*
 * The main routine reads words until end-of-file (´\0´ returned from "getchar")
 * "tree" is called to sort each word into the tree.
 */
main()
{
      struct tnode *top, *tree();
      char c, word[wsize];
      int i;

      i = top = 0;
      while (c=getchar())
            if (´a´<=c && c<=´z´ || ´A´<=c && c <=´Z´) {
                  if (i<wsize-1)
                        word[i++] = c;
            } else
                  if (i) {
                        word[i++] = ´\0´;
                        top = tree(top, word);
                        i = 0;
                  }
      tprint(top);
}
/*
 * The central routine.  If the subtree pointer is null, allocate a new node for it.
 * If the new word and the node´s word are the same, increase the node´s count.
 * Otherwise, recursively sort the word into the left or right subtree according
 * as the argument word is less or greater than the node´s word.
 */
struct tnode *tree(p, word)
struct tnode *p;
char word[];
{
      struct tnode *alloc();
      int cond;

      /* Is pointer null? */
      if (p==0) {
            p = alloc();
```

```
            copy(word, p->tword);
            p->count = 1;
            p->right = p->left = 0;
            return(p);
      }
      /* Is word repeated? */
      if ((cond=compar(p->tword, word)) == 0) {
            p->count++;
            return(p);
      }
      /* Sort into left or right */
      if (cond<0)
            p->left = tree(p->left, word);
      else
            p->right = tree(p->right, word);
      return(p);
}
/*
 * Print the tree by printing the left subtree, the given node, and the right subtre
 */
tprint(p)
struct tnode *p;
{
      while (p) {
            tprint(p->left);
            printf("%d:  %s\n", p->count, p->tword);
            p = p->right;
      }
}
/*
 * String comparison: return number (>, =, <) 0
 * according as s1 (>, =, <) s2.
 */
compar(s1, s2)
char *s1, *s2;
{
      int c1, c2;

      while((c1 = *s1++) == (c2 = *s2++))
            if (c1=='\0')
                  return(0);
      return(c2-c1);
}
/*
 * String copy: copy s1 into s2 until the null
 * character appears.
 */
copy(s1, s2)
char *s1, *s2;
{
      while(*s2++ = *s1++);
}
/*
 * Node allocation: return pointer to a free node.
 * Bomb out when all are gone.  Just for fun, there
 * is a mechanism for using nodes that have been
 * freed, even though no one here calls "free."
 */
struct tnode *alloc()
```

```
{
        struct tnode *t;

        if (freep) {
                t = freep;
                freep = freep->left;
                return(t);
        }
        if (--nnodes < 0) {
                printf("Out of space\n");
                exit();
        }
        return(spacep++);
}
/*
 * The uncalled routine which puts a node on the free list.
 */
free(p)
struct tnode *p;
{
        p->left = freep;
        freep = p;
}
```

To illustrate a slightly different technique of handling the same problem, we will repeat fragments of this example with the tree nodes treated explicitly as members of an array. The fundamental change is to deal with the subscript of the array member under discussion, instead of a pointer to it. The `struct` declaration becomes

```
struct tnode {
        char tword[wsize];
        int count;
        int left;
        int right;
};
```

and *alloc* becomes

```
alloc()
{
        int t;

        t = --nnodes;
        if (t<=0) {
                printf("Out of space\n");
                exit();
        }
        return(t);
}
```

The *free* stuff has disappeared because if we deal with exclusively with subscripts some sort of map has to be kept, which is too much trouble.

Now the *tree* routine returns a subscript also, and it becomes:

```
tree(p, word)
char word[];
{
        int cond;

        if (p==0) {
                p = alloc();
                copy(word, space[p].tword);
```

```
                space[p].count = 1;
                space[p].right = space[p].left = 0;
                return(p);
        }
        if ((cond=compar(space[p].tword, word)) == 0) {
                space[p].count++;
                return(p);
        }
        if (cond<0)
                space[p].left = tree(space[p].left, word);
        else
                space[p].right = tree(space[p].right, word);
        return(p);
}
```

The other routines are changed similarly. It must be pointed out that this version is noticeably less efficient than the first because of the multiplications which must be done to compute an offset in *space* corresponding to the subscripts.

The observation that subscripts (like ‘‘a[i]’’) are less efficient than pointer indirection (like ‘‘*ap’’) holds true independently of whether or not structures are involved. There are of course many situations where subscripts are indispensable, and others where the loss in efficiency is worth a gain in clarity.

16.3 Formatted output

Here is a simplified version of the *printf* routine, which is available in the C library. It accepts a string (character array) as first argument, and prints subsequent arguments according to specifications contained in this format string. Most characters in the string are simply copied to the output; two-character sequences beginning with ‘‘%’’ specify that the next argument should be printed in a style as follows:

%d	decimal number
%o	octal number
%c	ASCII character, or 2 characters if upper character is not null
%s	string (null-terminated array of characters)
%f	floating-point number

The actual parameters for each function call are laid out contiguously in increasing storage locations; therefore, a function with a variable number of arguments may take the address of (say) its first argument, and access the remaining arguments by use of subscripting (regarding the arguments as an array) or by indirection combined with pointer incrementation.

If in such a situation the arguments have mixed types, or if in general one wishes to insist that an lvalue should be treated as having a given type, then `struct` declarations like those illustrated below will be useful. It should be evident, though, that such techniques are implementation dependent.

Printf depends as well on the fact that `char` and `float` arguments are widened respectively to `int` and `double`, so there are effectively only two sizes of arguments to deal with. *Printf* calls the library routines *putchar* to write out single characters and *ftoa* to dispose of floating-point numbers.

```
printf(fmt, args)
char fmt[];
{
        char *s;
        struct { char **charpp; };
        struct { double *doublep; };
        int *ap, x, c;

        ap = &args;                   /* argument pointer */
        for ( ; ; ) {
                while((c = *fmt++) != ´%´) {
                        if(c == ´\0´)
                                return;
```

```
                        putchar(c);
                }
                switch (c = *fmt++) {
                /* decimal */
                case ´d´:
                        x = *ap++;
                        if(x < 0) {
                                x = -x;
                                if(x<0) {  /* is - infinity */
                                        printf("-32768");
                                        continue;
                                }
                                putchar(´-´);
                        }
                        printd(x);
                        continue;
                /* octal */
                case ´o´:
                        printo(*ap++);
                        continue;
                /* float, double */
                case ´f´:
                        /* let ftoa do the real work */
                        ftoa(*ap.doublep++);
                        continue;
                /* character */
                case ´c´:
                        putchar(*ap++);
                        continue;
                /* string */
                case ´s´:
                        s = *ap.charpp++;
                        while(c = *s++)
                                putchar(c);
                        continue;
                }
                putchar(c);
        }
}
/*
 * Print n in decimal; n must be non-negative
 */
printd(n)
{
        int a;
        if (a=n/10)
                printd(a);
        putchar(n%10 + ´0´);
}
/*
 * Print n in octal, with exactly 1 leading 0
 */
printo(n)
{
        if (n)
                printo((n>>3)&017777);
        putchar((n&07)+´0´);
}
```

REFERENCES

1. Johnson, S. C., and Kernighan, B. W. ''The Programming Language B.'' Comp. Sci. Tech. Rep. #8., Bell Laboratories, 1972.
2. Ritchie, D. M., and Thompson, K. L. ''The UNIX Time-sharing System.'' C. ACM 7, 17, July, 1974, pp. 365-375.
3. Peterson, T. G., and Lesk, M. E. ''A User's Guide to the C Language on the IBM 370.'' Internal Memorandum, Bell Laboratories, 1974.
4. Thompson, K. L., and Ritchie, D. M. *UNIX Programmer's Manual.* Bell Laboratories, 1973.
5. Lesk, M. E., and Barres, B. A. ''The GCOS C Library.'' Internal memorandum, Bell Laboratories, 1974.
6. Kernighan, B. W. ''Programming in C− A Tutorial.'' Unpublished internal memorandum, Bell Laboratories, 1974.

APPENDIX 1

Syntax Summary

1. Expressions.

expression:
primary
∗ expression
& expression
− expression
! expression
~expression
++ lvalue
−− lvalue
lvalue ++
lvalue −−
`sizeof` expression
expression binop expression
expression ? expression : expression
lvalue asgnop expression
expression , expression

primary:
identifier
constant
string
(expression)
primary (expression-list$_{opt}$)
primary [expression]
lvalue . identifier
primary > identifier

lvalue:
identifier
primary [expression]
lvalue . identifier
primary > identifier
∗ expression
(lvalue)

The primary-expression operators

`()   []   .   >`

have highest priority and group left-to-right. The unary operators

`&   −   !   ~   ++ −− sizeof`

have priority below the primary operators but higher than any binary operator, and group right-to-left. Binary operators and the conditional operator all group left-to-right, and have priority decreasing as indicated:

binop:
```
*    /    %
+    -
>>    <<
<    >    <=    >=
==    !=
&
```

```
^
|
&&
||
? :
```

Assignment operators all have the same priority, and all group right-to-left.

asgnop:

```
= =+ =- =* =/ =% =>> =<< =& =^ =|
```

The comma operator has the lowest priority, and groups left-to-right.

2. Declarations.

declaration:
  decl-specifiers declarator-list$_{opt}$ `;`

decl-specifiers:
  type-specifier
  sc-specifier
  type-specifier sc-specifier
  sc-specifier type-specifier

sc-specifier:
  `auto`
  `static`
  `extern`
  `register`

type-specifier:
  `int`
  `char`
  `float`
  `double`
  `struct` `{` *type-decl-list* `}`
  `struct` *identifier* `{` *type-decl-list* `}`
  `struct` *identifier*

declarator-list:
  declarator
  declarator `,` *declarator-list*

declarator:
  identifier
  `*` *declarator*
  declarator `( )`
  declarator `[` *constant-expression*$_{opt}$ `]`
  `(` *declarator* `)`

type-decl-list:
  type-declaration
  type-declaration type-decl-list

type-declaration:
  type-specifier declarator-list `;`

3. Statements.

statement:
  expression `;`
  `{` *statement-list* `}`

`if (` *expression* `)` *statement*
`if (` *expression* `)` *statement* `else` *statement*
`while (` *expression* `)` *statement*
`for (` *expression*$_{opt}$ `;` *expression*$_{opt}$ `;` *expression*$_{opt}$ `)` `statement`
`switch (` *expression* `)` *statement*
`case` *constant-expression* `:` *statement*
`default :` *statement*
`break ;`
`continue ;`
`return ;`
`return (` *expression* `) ;`
`goto` *expression* `;`
identifier `:` *statement*
`;`

statement-list:
statement
statement statement-list

4. External definitions.

program:
external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
type-specifier$_{opt}$ *function-declarator function-body*

function-declarator:
declarator `(` *parameter-list*$_{opt}$ `)`

parameter-list:
identifier
identifier `,` *parameter-list*

function-body:
type-decl-list function-statement

function-statement:
{ declaration-list$_{opt}$ *statement-list }*

data-definition:
`extern`$_{opt}$ *type-specifier*$_{opt}$ *init-declarator-list*$_{opt}$ `;`

init-declarator-list:
init-declarator
init-declarator `,` *init-declarator-list*

init-declarator:
declarator initializer$_{opt}$

initializer:
constant
{ constant-expression-list }

constant-expression-list:
constant-expression
constant-expression **,** *constant-expression-list*

constant-expression:
expression

5. Preprocessor

`# define` *identifier token-string*

`# include "`*filename*`"`

APPENDIX 2

Implementation Peculiarities

This Appendix briefly summarizes the differences between the implementations of C on the PDP-11 under UNIX and on the HIS 6070 under GCOS; it includes some known bugs in each implementation. Each entry is keyed by an indicator as follows:

h	hard to fix
g	GCOS version should probably be changed
u	UNIX version should probably be changed
d	Inherent difference likely to remain

This list was prepared by M. E. Lesk, S. C. Johnson, E. N. Pinson, and the author.

A. Bugs or differences from C language specifications

hg	A.1)	GCOS does not do type conversions in ‘‘?:’’.
hg	A.2)	GCOS has a bug in `int` and `real` comparisons; the numbers are compared by subtraction, and the difference must not overflow.
g	A.3)	When *x* is a `float`, the construction ‘‘test ? −x : x’’ is illegal on GCOS.
hg	A.4)	‘‘p1−>p2 =+ 2’’ causes a compiler error, where p1 and p2 are pointers.
u	A.5)	On UNIX, the expression in a `return` statement is *not* converted to the type of the function, as promised.
hug	A.6)	`entry` statement is not implemented at all.

B. Implementation differences

d	B.1)	Sizes of character constants differ; UNIX: 2, GCOS: 4.
d	B.2)	Table sizes in compilers differ.
d	B.3)	`chars` and `ints` have different sizes; `chars` are 8 bits on UNIX, 9 on GCOS; words are 16 bits on UNIX and 36 on GCOS. There are corresponding differences in representations of `floats` and `doubles`.
d	B.4)	Character arrays stored left to right in a word in GCOS, right to left in UNIX.
g	B.5)	Passing of floats and doubles differs; UNIX passes on stack, GCOS passes pointer (hidden to normal user).
g	B.6)	Structures and strings are aligned on a word boundary in UNIX, not aligned in GCOS.
g	B.7)	GCOS preprocessor supports #rename, #escape; UNIX has only #define, #include.
u	B.8)	Preprocessor is not invoked on UNIX unless first character of file is ‘‘#’’.
u	B.9)	The external definition ‘‘static int ...’’ is legal on GCOS, but gets a diagnostic on UNIX. (On GCOS it means an identifier global to the routines in the file but invisible to routines compiled separately.)
g	B.10)	A compound statement on GCOS must contain one ‘‘;’’ but on UNIX may be empty.
g	B.11)	On GCOS case distinctions in identifiers and keywords are ignored; on UNIX case is significant everywhere, with keywords in lower case.

C. Syntax Differences

g	C.1)	UNIX allows broader classes of initialization; on GCOS an initializer must be a constant, name, or string. Similarly, GCOS is much stickier about wanting braces around initializers and in particular they must be present for array initialization.
g	C.2)	‘‘int extern’’ illegal on GCOS; must have ‘‘extern int’’ (storage class before type).
g	C.3)	Externals on GCOS must have a type (not defaulted to `int`).
u	C.4)	GCOS allows initialization of internal `static` (same syntax as for external definitions).
g	C.5)	integer−>... is not allowed on GCOS.
g	C.6)	Some operators on pointers are illegal on GCOS (<, >).

g	C.7)	`register` storage class means something on UNIX, but is not accepted on GCOS.
g	C.8)	Scope holes: ‘‘int x; f ( ) {int x;}’’ is illegal on UNIX but defines two variables on GCOS.
g	C.9)	When function names are used as arguments on UNIX, either ‘‘fname’’ or ‘‘&fname’’ may be used to get a pointer to the function; on GCOS ‘‘&fname’’ generates a doubly-indirect pointer. (Note that both are wrong since the ‘‘&’’ is supposed to be supplied for free.)

D. Operating System Dependencies

d	D.1)	GCOS allocates external scalars by SYMREF; UNIX allocates external scalars as labelled common; as a result there may be many uninitialized external definitions of the same variable on UNIX but only one on GCOS.
d	D.2)	External names differ in allowable length and character set; on UNIX, 7 characters and both cases; on GCOS 6 characters and only one case.

E. Semantic Differences

hg	E.1)	‘‘int i, *p; p=i; i=p;’’ does nothing on UNIX, does something on GCOS (destroys right half of i) .
d	E.2)	‘‘>>’’ means arithmetic shift on UNIX, logical on GCOS.
d	E.3)	When a `char` is converted to integer, the result is always positive on GCOS but can be negative on UNIX.
d	E.4)	Arguments of subroutines are evaluated left-to-right on GCOS, right-to-left on UNIX.

Lint, a C Program Checker

S. C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Lint is a command which examines C source programs, detecting a number of bugs and obscurities. It enforces the type rules of C more strictly than the C compilers. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful, or error prone, constructions which nevertheless are, strictly speaking, legal.

Lint accepts multiple input files and library specifications, and checks them for consistency.

The separation of function between *lint* and the C compilers has both historical and practical rationale. The compilers turn C programs into executable files rapidly and efficiently. This is possible in part because the compilers do not do sophisticated type checking, especially between separately compiled programs. *Lint* takes a more global, leisurely view of the program, looking much more carefully at the compatibilities.

This document discusses the use of *lint*, gives an overview of the implementation, and gives some hints on the writing of machine independent C code.

July 26, 1978

Lint, a C Program Checker

S. C. Johnson

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction and Usage

Suppose there are two C[1] source files, *file1.* c and *file2.c* , which are ordinarily compiled and loaded together. Then the command

lint file1.c file2.c

produces messages describing inconsistencies and inefficiencies in the programs. The program enforces the typing rules of C more strictly than the C compilers (for both historical and practical reasons) enforce them. The command

lint −p file1.c file2.c

will produce, in addition to the above messages, additional messages which relate to the portability of the programs to other operating systems and machines. Replacing the **−p** by **−h** will produce messages about various error-prone or wasteful constructions which, strictly speaking, are not bugs. Saying **−hp** gets the whole works.

The next several sections describe the major messages; the document closes with sections discussing the implementation and giving suggestions for writing portable C. An appendix gives a summary of the *lint* options.

A Word About Philosophy

Many of the facts which *lint* needs may be impossible to discover. For example, whether a given function in a program ever gets called may depend on the input data. Deciding whether *exit* is ever called is equivalent to solving the famous ‘‘halting problem,’’ known to be recursively undecidable.

Thus, most of the *lint* algorithms are a compromise. If a function is never mentioned, it can never be called. If a function is mentioned, *lint* assumes it can be called; this is not necessarily so, but in practice is quite reasonable.

Lint tries to give information with a high degree of relevance. Messages of the form ‘‘*xxx* might be a bug’’ are easy to generate, but are acceptable only in proportion to the fraction of real bugs they uncover. If this fraction of real bugs is too small, the messages lose their credibility and serve merely to clutter up the output, obscuring the more important messages.

Keeping these issues in mind, we now consider in more detail the classes of messages which *lint* produces.

Unused Variables and Functions

As sets of programs evolve and develop, previously used variables and arguments to functions may become unused; it is not uncommon for external variables, or even entire functions, to become unnecessary, and yet not be removed from the source. These ‘‘errors of commission’’ rarely cause working programs to fail, but they are a source of inefficiency, and make programs harder to understand and change. Moreover, information about such unused variables and functions can occasionally serve to discover bugs; if a function does a necessary job, and is never called, something is wrong!

Lint complains about variables and functions which are defined but not otherwise mentioned. An exception is variables which are declared through explicit **extern** statements but are never referenced;

thus the statement

```
extern float sin( );
```

will evoke no comment if *sin* is never used. Note that this agrees with the semantics of the C compiler. In some cases, these unused external declarations might be of some interest; they can be discovered by adding the −**x** flag to the *lint* invocation.

Certain styles of programming require many functions to be written with similar interfaces; frequently, some of the arguments may be unused in many of the calls. The −**v** option is available to suppress the printing of complaints about unused arguments. When −**v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments; this can be considered an active (and preventable) waste of the register resources of the machine.

There is one case where information about unused, or undefined, variables is more distracting than helpful. This is when *lint* is applied to some, but not all, files out of a collection which are to be loaded together. In this case, many of the functions and variables defined may not be used, and, conversely, many functions and variables defined elsewhere may be used. The −**u** flag may be used to suppress the spurious messages which might otherwise appear.

Set/Used Information

Lint attempts to detect cases where a variable is used before it is set. This is very difficult to do well; many algorithms take a good deal of time and space, and still produce messages about perfectly valid programs. *Lint* detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. It assumes that taking the address of a variable constitutes a ''use,'' since the actual use may occur at any later time, in a data dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement, since the true flow of control need not be discovered. It does mean that *lint* can complain about some programs which are legal, but these programs would probably be considered bad on stylistic grounds (e.g. might contain at least two **goto**'s). Because static and external variables are initialized to 0, no meaningful information can be discovered about their uses. The algorithm deals correctly, however, with initialized automatic variables, and variables which are used in the expression which first sets them.

The set/used information also permits recognition of those local variables which are set and never used; these form a frequent source of inefficiencies, and may also be symptomatic of bugs.

Flow of Control

Lint attempts to detect unreachable portions of the programs which it processes. It will complain about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. An attempt is made to detect loops which can never be left at the bottom, detecting the special cases **while**(1) and **for**(;;) as infinite loops. *Lint* also complains about loops which cannot be entered at the top; some valid programs may have such loops, but at best they are bad style, at worst bugs.

Lint has an important area of blindness in the flow of control algorithm: it has no way of detecting functions which are called and never return. Thus, a call to *exit* may cause unreachable code which *lint* does not detect; the most serious effects of this are in the determination of returned function values (see the next section).

One form of unreachable statement is not usually complained about by *lint;* a **break** statement that cannot be reached causes no message. Programs generated by *yacc*,[2] and especially *lex*,[3] may have literally hundreds of unreachable **break** statements. The −**O** flag in the C compiler will often eliminate the resulting object code inefficiency. Thus, these unreached statements are of little importance, there is typically nothing the user can do about them, and the resulting messages would clutter up the *lint* output. If these messages are desired, *lint* can be invoked with the −**b** option.

Function Values

Sometimes functions return values which are never used; sometimes programs incorrectly use function ''values'' which have never been returned. *Lint* addresses this problem in a number of ways.

Locally, within a function definition, the appearance of both

```
return( expr );
```

and

```
return ;
```

statements is cause for alarm; *lint* will give the message

function *name* contains return(e) and return

The most serious difficulty with this is detecting when a function return is implied by flow of control reaching the end of the function. This can be seen with a simple example:

```
f ( a ) {
        if ( a ) return ( 3 );
        g ();
        }
```

Notice that, if *a* tests false, *f* will call *g* and then return with no defined return value; this will trigger a complaint from *lint*. If *g*, like *exit*, never returns, the message will still be produced when in fact nothing is wrong.

In practice, some potentially serious bugs have been discovered by this feature; it also accounts for a substantial fraction of the ''noise'' messages produced by *lint*.

On a global scale, *lint* detects cases where a function returns a value, but this value is sometimes, or always, unused. When the value is always unused, it may constitute an inefficiency in the function definition. When the value is sometimes unused, it may represent bad style (e.g., not testing for error conditions).

The dual problem, using a function value when the function does not return one, is also detected. This is a serious problem. Amazingly, this bug has been observed on a couple of occasions in ''working'' programs; the desired function value just happened to have been computed in the function return register!

Type Checking

Lint enforces the type checking rules of C more strictly than the compilers do. The additional checking is in four major areas: across certain binary operators and implied assignments, at the structure selection operators, between the definition and uses of functions, and in the use of enumerations.

There are a number of operators which have an implied balancing between types of the operands. The assignment, conditional (?:), and relational operators have this property; the argument of a **return** statement, and expressions used in initialization also suffer similar conversions. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly, except that arrays of *x*'s can, of course, be intermixed with pointers to *x*'s.

The type checking rules also require that, in structure references, the left operand of the —> be a pointer to structure, the left operand of the **.** be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

With enumerations, checks are made that enumeration variables or members are not mixed with other types, or other enumerations, and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

Type Casts

The type cast feature in C was introduced largely as an aid to producing more portable programs. Consider the assignment

```
p = 1 ;
```

where *p* is a character pointer. *Lint* will quite rightly complain. Now, consider the assignment

```
p = (char *)1 ;
```

in which a cast has been used to convert the integer to a character pointer. The programmer obviously had a strong motivation for doing this, and has clearly signaled his intentions. It seems harsh for *lint* to continue to complain about this. On the other hand, if this code is moved to another machine, such code should be looked at carefully. The −**c** flag controls the printing of comments about casts. When −**c** is in effect, casts are treated as though they were assignments subject to complaint; otherwise, all legal casts are passed without comment, no matter how strange the type mixing seems to be.

Nonportable Character Use

On the PDP-11, characters are signed quantities, with a range from −128 to 127. On most of the other C implementations, characters take on only positive values. Thus, *lint* will flag certain comparisons and assignments as being illegal or nonportable. For example, the fragment

```
char c;
        ...
if( (c = getchar( )) < 0 ) ....
```

works on the PDP-11, but will fail on machines where characters always take on positive values. The real solution is to declare *c* an integer, since *getchar* is actually returning integer values. In any case, *lint* will say ‘‘nonportable character comparison’’.

A similar issue arises with bitfields; when assignments of constant values are made to bitfields, the field may be too small to hold the value. This is especially true because on some machines bitfields are considered as signed quantities. While it may seem unintuitive to consider that a two bit field declared of type **int** cannot hold the value 3, the problem disappears if the bitfield is declared to have type **unsigned**.

Assignments of longs to ints

Bugs may arise from the assignment of **long** to an **int**, which loses accuracy. This may happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program can stop working because some intermediate results may be assigned to **ints**, losing accuracy. Since there are a number of legitimate reasons for assigning **longs** to **ints**, the detection of these assignments is enabled by the −**a** flag.

Strange Constructions

Several perfectly legal, but somewhat strange, constructions are flagged by *lint;* the messages hopefully encourage better code quality, clearer style, and may even point out bugs. The −**h** flag is used to enable these checks. For example, in the statement

```
*p++ ;
```

the * does nothing; this provokes the message ‘‘null effect’’ from *lint*. The program fragment

```
unsigned x ;
if( x < 0 ) ...
```

is clearly somewhat strange; the test will never succeed. Similarly, the test

```
if( x > 0 ) ...
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. *Lint* will say ‘‘degenerate unsigned comparison’’ in these cases. If one says

```
if( 1 != 0 ) ....
```

lint will report ‘‘constant in conditional context’’, since the comparison of 1 with 0 gives a constant result.

Another construction detected by *lint* involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs extremely hard to find. For example, the statements

```
if( x&077 == 0 ) ...
```

or

```
x≪2 + 40
```

probably do not do what was intended. The best solution is to parenthesize such expressions, and *lint* encourages this by an appropriate message.

Finally, when the −**h** flag is in force *lint* complains about variables which are redeclared in inner blocks in a way that conflicts with their use in outer blocks. This is legal, but is considered by many (including the author) to be bad style, usually unnecessary, and frequently a bug.

Ancient History

There are several forms of older syntax which are being officially discouraged. These fall into two classes, assignment operators and initialization.

The older forms of assignment operators (e.g., =+, =–, . . .) could cause ambiguous expressions, such as

```
a  =–1 ;
```

which could be taken as either

```
a =–  1 ;
```

or

```
a  =  −1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer, and preferred operators (+=, −=, etc.) have no such ambiguities. To spur the abandonment of the older forms, *lint* complains about these old fashioned operators.

A similar issue arises with initialization. The older language allowed

```
int  x  1 ;
```

to initialize *x* to 1. This also caused syntactic difficulties: for example,

```
int  x  ( −1 ) ;
```

looks somewhat like the beginning of a function declaration:

```
int  x  ( y ) {  . . .
```

and the compiler must read a fair ways past *x* in order to sure what the declaration really is.. Again, the problem is even more perplexing when the initializer involves a macro. The current syntax places an equals sign between the variable and the initializer:

int x = −1 ;

This is free of any possible syntactic ambiguity.

Pointer Alignment

Certain pointer assignments may be reasonable on some machines, and illegal on others, due entirely to alignment restrictions. For example, on the PDP-11, it is reasonable to assign integer pointers to double pointers, since double precision values may begin on any integer boundary. On the Honeywell 6000, double precision values must begin on even word boundaries; thus, not all such assignments make sense. *Lint* tries to detect cases where pointers are assigned to other pointers, and such alignment problems might arise. The message ‘‘possible pointer alignment problem’’ results from this situation whenever either the −**p** or −**h** flags are in effect.

Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines (like the PDP-11) in which the stack runs backwards, function arguments will probably be best evaluated from right-to-left; on machines with a stack running forward, left-to-right seems most attractive. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators which have side effects, such as the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be unduly compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler, and, in fact, the various C compilers have considerable differences in the order in which they will evaluate complicated expressions. In particular, if any variable is changed by a side effect, and also used elsewhere in the same expression, the result is explicitly undefined.

Lint checks for the important special case where a simple scalar variable is affected. For example, the statement

a[*i*] = *b*[*i*++] ;

will draw the complaint:

warning: *i* evaluation order undefined

Implementation

Lint consists of two programs and a driver. The first program is a version of the Portable C Compiler[45] which is the basis of the IBM 370, Honeywell 6000, and Interdata 8/32 C compilers. This compiler does lexical and syntax analysis on the input text, constructs and maintains symbol tables, and builds trees for expressions. Instead of writing an intermediate file which is passed to a code generator, as the other compilers do, *lint* produces an intermediate file which consists of lines of ascii text. Each line contains an external variable name, an encoding of the context in which it was seen (use, definition, declaration, etc.), a type specifier, and a source file name and line number. The information about variables local to a function or file is collected by accessing the symbol table, and examining the expression trees.

Comments about local problems are produced as detected. The information about external names is collected onto an intermediate file. After all the source files and library descriptions have been collected, the intermediate file is sorted to bring all information collected about a given external name together. The second, rather small, program then reads the lines from the intermediate file and compares all of the definitions, declarations, and uses for consistency.

The driver controls this process, and is also responsible for making the options available to both passes of *lint*.

Portability

C on the Honeywell and IBM systems is used, in part, to write system code for the host operating system. This means that the implementation of C tends to follow local conventions rather than adhere strictly to UNIX† system conventions. Despite these differences, many C programs have been successfully moved to GCOS and the various IBM installations with little effort. This section describes some of the differences between the implementations, and discusses the *lint* features which encourage portability.

Uninitialized external variables are treated differently in different implementations of C. Suppose two files both contain a declaration without initialization, such as

```
int a ;
```

outside of any function. The UNIX loader will resolve these declarations, and cause only a single word of storage to be set aside for *a*. Under the GCOS and IBM implementations, this is not feasible (for various stupid reasons!) so each such declaration causes a word of storage to be set aside and called *a*. When loading or library editing takes place, this causes fatal conflicts which prevent the proper operation of the program. If *lint* is invoked with the **−p** flag, it will detect such multiple definitions.

A related difficulty comes from the amount of information retained about external names during the loading process. On the UNIX system, externally known names have seven significant characters, with the upper/lower case distinction kept. On the IBM systems, there are eight significant characters, but the case distinction is lost. On GCOS, there are only six characters, of a single case. This leads to situations where programs run on the UNIX system, but encounter loader problems on the IBM or GCOS systems. *Lint* **−p** causes all external symbols to be mapped to one case and truncated to six characters, providing a worst-case analysis.

A number of differences arise in the area of character handling: characters in the UNIX system are eight bit ascii, while they are eight bit ebcdic on the IBM, and nine bit ascii on GCOS. Moreover, character strings go from high to low bit positions (‘‘left to right’’) on GCOS and IBM, and low to high (‘‘right to left’’) on the PDP-11. This means that code attempting to construct strings out of character constants, or attempting to use characters as indices into arrays, must be looked at with great suspicion. *Lint* is of little help here, except to flag multi-character character constants.

Of course, the word sizes are different! This causes less trouble than might be expected, at least when moving from the UNIX system (16 bit words) to the IBM (32 bits) or GCOS (36 bits). The main problems are likely to arise in shifting or masking. C now supports a bit-field facility, which can be used to write much of this code in a reasonably portable way. Frequently, portability of such code can be enhanced by slight rearrangements in coding style. Many of the incompatibilities seem to have the flavor of writing

```
x &= 0177700 ;
```

to clear the low order six bits of *x*. This suffices on the PDP-11, but fails badly on GCOS and IBM. If the bit field feature cannot be used, the same effect can be obtained by writing

```
x &= ~ 077 ;
```

which will work on all these machines.

The right shift operator is arithmetic shift on the PDP-11, and logical shift on most other machines. To obtain a logical shift on all machines, the left operand can be typed **unsigned**. Characters are considered signed integers on the PDP-11, and unsigned on the other machines. This persistence of the sign bit may be reasonably considered a bug in the PDP-11 hardware which has infiltrated itself into the C language. If there were a good way to discover the programs which would be affected, C could be changed; in any case, *lint* is no help here.

The above discussion may have made the problem of portability seem bigger than it in fact is. The issues involved here are rarely subtle or mysterious, at least to the implementor of the program,

†UNIX is a Trademark of Bell Laboratories.

although they can involve some work to straighten out. The most serious bar to the portability of UNIX system utilities has been the inability to mimic essential UNIX system functions on the other systems. The inability to seek to a random character position in a text file, or to establish a pipe between processes, has involved far more rewriting and debugging than any of the differences in C compilers. On the other hand, *lint* has been very helpful in moving the UNIX operating system and associated utility programs to other machines.

Shutting Lint Up

There are occasions when the programmer is smarter than *lint*. There may be valid reasons for ''illegal'' type casts, functions with a variable number of arguments, etc. Moreover, as specified above, the flow of control information produced by *lint* often has blind spots, causing occasional spurious messages about perfectly reasonable programs. Thus, some way of communicating with *lint*, typically to shut it up, is desirable.

The form which this mechanism should take is not at all clear. New keywords would require current and old compilers to recognize these keywords, if only to ignore them. This has both philosophical and practical problems. New preprocessor syntax suffers from similar problems.

What was finally done was to cause a number of words to be recognized by *lint* when they were embedded in comments. This required minimal preprocessor changes; the preprocessor just had to agree to pass comments through to its output, instead of deleting them as had been previously done. Thus, *lint* directives are invisible to the compilers, and the effect on systems with the older preprocessors is merely that the *lint* directives don't work.

The first directive is concerned with flow of control information; if a particular place in the program cannot be reached, but this is not apparent to *lint*, this can be asserted by the directive

```
/* NOTREACHED */
```

at the appropriate spot in the program. Similarly, if it is desired to turn off strict type checking for the next expression, the directive

```
/* NOSTRICT */
```

can be used; the situation reverts to the previous default after the next expression. The **−v** flag can be turned on for one function by the directive

```
/* ARGSUSED */
```

Complaints about variable number of arguments in calls to a function can be turned off by the directive

```
/* VARARGS */
```

preceding the function definition. In some cases, it is desirable to check the first several arguments, and leave the later arguments unchecked. This can be done by following the VARARGS keyword immediately with a digit giving the number of arguments which should be checked; thus,

```
/* VARARGS2 */
```

will cause the first two arguments to be checked, the others unchecked. Finally, the directive

```
/* LINTLIBRARY */
```

at the head of a file identifies this file as a library declaration file; this topic is worth a section by itself.

Library Declaration Files

Lint accepts certain library directives, such as

```
-ly
```

and tests the source files for compatibility with these libraries. This is done by accessing library description files whose names are constructed from the library directives. These files all begin with the directive

```
/* LINTLIBRARY */
```

which is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. The VARARGS and ARGSUSED directives can be used to specify features of the library functions.

Lint library files are processed almost exactly like ordinary source files. The only difference is that functions which are defined on a library file, but are not used on a source file, draw no complaints. *Lint* does not simulate a full library search algorithm, and complains if the source files contain a redefinition of a library routine (this is a feature!).

By default, *lint* checks the programs it is given against a standard library file, which contains descriptions of the programs which are normally loaded when a C program is run. When the **-p** flag is in effect, another file is checked containing descriptions of the standard I/O library routines which are expected to be portable across various machines. The **-n** flag can be used to suppress all library checking.

Bugs, etc.

Lint was a difficult program to write, partially because it is closely connected with matters of programming style, and partially because users usually don't notice bugs which cause *lint* to miss errors which it should have caught. (By contrast, if *lint* incorrectly complains about something that is correct, the programmer reports that immediately!)

A number of areas remain to be further developed. The checking of structures and arrays is rather inadequate; size incompatibilities go unchecked, and no attempt is made to match up structure and union declarations across files. Some stricter checking of the use of the **typedef** is clearly desirable, but what checking is appropriate, and how to carry it out, is still to be determined.

Lint shares the preprocessor with the C compiler. At some point it may be appropriate for a special version of the preprocessor to be constructed which checks for things such as unused macro definitions, macro arguments which have side effects which are not expanded at all, or are expanded more than once, etc.

The central problem with *lint* is the packaging of the information which it collects. There are many options which serve only to turn off, or slightly modify, certain features. There are pressures to add even more of these options.

In conclusion, it appears that the general notion of having two programs is a good one. The compiler concentrates on quickly and accurately turning the program text into bits which can be run; *lint* concentrates on issues of portability, style, and efficiency. *Lint* can afford to be wrong, since incorrectness and over-conservatism are merely annoying, not fatal. The compiler can be fast since it knows that *lint* will cover its flanks. Finally, the programmer can concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of *lint*, the desirable properties of universality and portability.

References

1. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey (1978).
2. S. C. Johnson, ''Yacc — Yet Another Compiler-Compiler,'' Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).
3. M. E. Lesk, ''Lex — A Lexical Analyzer Generator,'' Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, Murray Hill, New Jersey (October 1975).
4. S. C. Johnson and D. M. Ritchie, ''UNIX Time-Sharing System: Portability of C Programs and the UNIX System,'' *Bell Sys. Tech. J.* **57**(6), pp.2021-2048 (1978).
5. S. C. Johnson, ''A Portable Compiler: Theory and Practice,'' *Proc. 5th ACM Symp. on Principles of Programming Languages*, pp.97-104 (January 1978).

Appendix: Current Lint Options

The command currently has the form

lint [−options] files... library-descriptors...

The options are

h Perform heuristic checks

p Perform portability checks

v Don't report unused arguments

u Don't report unused or undefined externals

b Report unreachable **break** statements.

x Report unused external declarations

a Report assignments of **long** to **int** or shorter.

c Complain about questionable casts

n No library checking is done

s Same as **h** (for historical reasons)

Make — A Program for Maintaining Computer Programs

S. I. Feldman

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

In a programming project, it is easy to lose track of which files need to be reprocessed or recompiled after a change is made in some part of the source. *Make* provides a simple mechanism for maintaining up-to-date versions of programs that result from many operations on a number of files. It is possible to tell *Make* the sequence of commands that create certain files, and the list of files that require other files to be current before the operations can be done. Whenever a change is made in any part of the program, the *Make* command will create the proper files simply, correctly, and with a minimum amount of effort.

The basic operation of *Make* is to find the name of a needed target in the description, ensure that all of the files on which it depends exist and are up to date, and then create the target if it has not been modified since its generators were. The description file really defines the graph of dependencies; *Make* does a depth-first search of this graph to determine what work is really necessary.

Make also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

August 15, 1978

Make — A Program for Maintaining Computer Programs

S. I. Feldman

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

It is common practice to divide large programs into smaller, more manageable pieces. The pieces may require quite different treatments: some may need to be run through a macro processor, some may need to be processed by a sophisticated program generator (e.g., Yacc[1] or Lex[2]). The outputs of these generators may then have to be compiled with special options and with certain definitions and declarations. The code resulting from these transformations may then need to be loaded together with certain libraries under the control of special options. Related maintenance activities involve running complicated test scripts and installing validated modules. Unfortunately, it is very easy for a programmer to forget which files depend on which others, which files have been modified recently, and the exact sequence of operations needed to make or exercise a new version of the program. After a long editing session, one may easily lose track of which files have been changed and which object modules are still valid, since a change to a declaration can obsolete a dozen other files. Forgetting to compile a routine that has been changed or that uses changed declarations will result in a program that will not work, and a bug that can be very hard to track down. On the other hand, recompiling everything in sight just to be safe is very wasteful.

The program described in this report mechanizes many of the activities of program development and maintenance. If the information on inter-file dependences and command sequences is stored in a file, the simple command

```
make
```

is frequently sufficient to update the interesting files, regardless of the number that have been edited since the last ''make''. In most cases, the description file is easy to write and changes infrequently. It is usually easier to type the *make* command than to issue even one of the needed operations, so the typical cycle of program development operations becomes

think — edit — *make* — test . . .

Make is most useful for medium-sized programming projects; it does not solve the problems of maintaining multiple source versions or of describing huge programs. *Make* was designed for use on Unix, but a version runs on GCOS.

Basic Features

The basic operation of *make* is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. *Make* does a depth-first search of the graph of dependences. The operation of the command depends on the ability to find the date and time that a file was last modified.

To illustrate, let us consider a simple example: A program named *prog* is made by compiling and loading three C-language files *x.c*, *y.c*, and *z.c* with the *lS* library. By convention, the output of the C compilations will be found in files named *x.o*, *y.o*, and *z.o*. Assume that the files *x.c* and *y.c* share some declarations in a file named *defs*, but that *z.c* does not. That is, *x.c* and *y.c* have the line

```
#include "defs"
```

The following text describes the relationships and operations:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  −lS  −o  prog

x.o  y.o :  defs
```

If this information were stored in a file named *makefile*, the command

```
make
```

would perform the operations needed to recreate *prog* after any changes had been made to any of the four source files *x.c*, *y.c*, *z.c*, or *defs*.

Make operates using three sources of information: a user-supplied description file (as above), file names and ''last-modified'' times from the file system, and built-in rules to bridge some of the gaps. In our example, the first line says that *prog* depends on three ''*.o*'' files. Once these object files are current, the second line describes how to load them to create *prog*. The third line says that *x.o* and *y.o* depend on the file *defs*. From the file system, *make* discovers that there are three ''*.c*'' files corresponding to the needed ''*.o*'' files, and uses built-in information on how to generate an object from a source file (*i.e.,* issue a ''cc −c'' command).

The following long-winded description file is equivalent to the one above, but takes no advantage of *make*'s innate knowledge:

```
prog :  x.o  y.o  z.o
        cc  x.o  y.o  z.o  −lS  −o  prog
x.o :  x.c  defs
        cc  −c  x.c
y.o :  y.c  defs
        cc  −c  y.c
z.o :  z.c
        cc  −c  z.c
```

If none of the source or object files had changed since the last time *prog* was made, all of the files would be current, and the command

```
make
```

would just announce this fact and stop. If, however, the *defs* file had been edited, *x.c* and *y.c* (but not *z.c*) would be recompiled, and then *prog* would be created from the new ''*.o*'' files. If only the file *y.c* had changed, only it would be recompiled, but it would still be necessary to reload *prog*.

If no target name is given on the *make* command line, the first target mentioned in the description is created; otherwise the specified targets are made. The command

```
make x.o
```

would recompile *x.o* if *x.c* or *defs* had changed.

If the file exists after the commands are executed, its time of last modification is used in further decisions; otherwise the current time is used. It is often quite useful to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of *make*'s ability to generate files and substitute macros. Thus, an entry ''save'' might be included to copy a certain set of files, or an entry ''cleanup'' might be used to throw away unneeded intermediate files. In other cases one may maintain a zero-length file purely to keep track of the time at which certain actions were performed. This technique is useful for maintaining remote archives and listings.

Make has a simple macro mechanism for substituting in dependency lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be parenthesized. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two invocations are identical. $$ is a dollar sign. All of these macros are assigned values during input, as shown below. Four special macros change values during the execution of the command: $*, $@, $?, and $<. They will be discussed later. The following fragment shows the use:

```
OBJECTS = x.o y.o z.o
LIBES = −lS
prog: $(OBJECTS)
        cc $(OBJECTS)  $(LIBES)  −o prog
  . . .
```

The command

```
make
```

loads the three object files with the *lS* library. The command

```
make  "LIBES= −ll −lS"
```

loads them with both the Lex (‘‘−ll’’) and the Standard (‘‘−lS’’) libraries, since macro definitions on the command line override definitions in the description. (It is necessary to quote arguments with embedded blanks in UNIX† commands.)

The following sections detail the form of description files and the command line, and discuss options and built-in rules in more detail.

Description Files and Substitutions

A description file contains three types of information: macro definitions, dependency information, and executable commands. There is also a comment convention: all characters after a sharp (#) are ignored, as is the sharp itself. Blank lines and lines beginning with a sharp are totally ignored. If a non-comment line is too long, it can be continued using a backslash. If the last character of a line is a backslash, the backslash, newline, and following blanks and tabs are replaced by a single blank.

A macro definition is a line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped.) The following are valid macro definitions:

```
2 = xyz
abc = −ll −ly −lS
LIBES =
```

The last definition assigns LIBES the null string. A macro that is never explicitly defined has the null string as value. Macro definitions may also appear on the *make* command line (see below).

Other lines give information about target files. The general form of an entry is:

target1 [target2 . . .] :[:] [dependent1 . . .] [; commands] [# . . .]
[*(tab)* commands] [# . . .]
 . . .

Items inside brackets may be omitted. Targets and dependents are strings of letters, digits, periods, and slashes. (Shell metacharacters ‘‘*’’ and ‘‘?’’ are expanded.) A command is any string of characters not including a sharp (except in quotes) or newline. Commands may appear either after a semicolon on a

†UNIX is a Trademark of Bell Laboratories.

dependency line or on lines beginning with a tab immediately following a dependency line.

A dependency line may have either a single or a double colon. A target name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

1. For the usual single-colon case, at most one of these dependency lines may have a command sequence associated with it. If the target is out of date with any of the dependents on any of the lines, and a command sequence is specified (even a null one following a semicolon or tab), it is executed; otherwise a default creation rule may be invoked.

2. In the double-colon case, a command sequence may be associated with each dependency line; if the target is out of date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. This detailed form is of particular value in updating archive-type files.

If a target must be created, the sequence of commands is executed. Normally, each command line is printed and then passed to a separate invocation of the Shell after substituting for macros. (The printing is suppressed in silent mode or if the command line begins with an @ sign). *Make* normally stops if any command signals an error by returning a non-zero error code. (Errors are ignored if the ''−i'' flags has been specified on the *make* command line, if the fake target name ''.IGNORE'' appears in the description file, or if the command string in the description file begins with a hyphen. Some UNIX commands return meaningless status). Because each command line is passed to a separate invocation of the Shell, care must be taken with certain commands (e.g., *cd* and Shell control commands) that have meaning only within a single Shell process; the results are forgotten before the next line is executed.

Before issuing any command, certain macros are set. $@ is set to the name of the file to be ''made''. $? is set to the string of names that were found to be younger than the target. If the command was generated by an implicit rule (see below), $< is the name of the related file that caused the action, and $* is the prefix shared by the current and the dependent file names.

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name ''.DEFAULT'' are used. If there is no such name, *make* prints a message and stops.

Command Usage

The *make* command takes four kinds of arguments: macro definitions, flags, description file names, and target file names.

make [flags] [macro definitions] [targets]

The following summary of the operation of the command explains how these arguments are interpreted.

First, all macro definition arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.

Next, the flag arguments are examined. The permissible flags are

−i Ignore error codes returned by invoked commands. This mode is entered if the fake target name ''.IGNORE'' appears in the description file.

−s Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name ''.SILENT'' appears in the description file.

−r Do not use the built-in rules.

−n No execute mode. Print commands, but do not execute them. Even lines beginning with an ''@'' sign are printed.

−t Touch the target files (causing them to be up to date) rather than issue the usual commands.

−q Question. The *make* command returns a zero or non-zero status code depending on whether the target file is or is not up to date.

–p Print out the complete set of macro definitions and target descriptions

–d Debug mode. Print out detailed information on files and times examined.

–f Description file name. The next argument is assumed to be the name of a description file. A file name of ‘‘–’’ denotes the standard input. If there are no ‘‘–f’’ arguments, the file named *makefile* or *Makefile* in the current directory is read. The contents of the description files override the built-in rules if they are present).

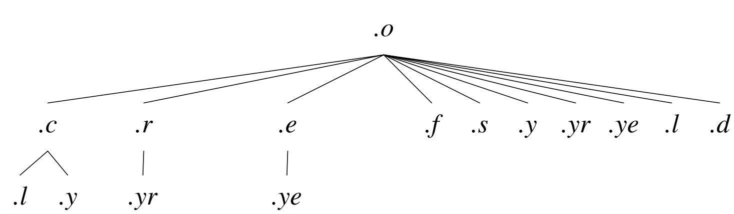
Finally, the remaining arguments are assumed to be the names of targets to be made; they are done in left to right order. If there are no such arguments, the first name in the description files that does not begin with a period is ‘‘made’’.

Implicit Rules

The *make* program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands. (The Appendix describes these tables and means of overriding them.) The default suffix list is:

.o	Object file
.c	C source file
.e	Efl source file
.r	Ratfor source file
.f	Fortran source file
.s	Assembler source file
.y	Yacc-C source grammar
.yr	Yacc-Ratfor source grammar
.ye	Yacc-Efl source grammar
.l	Lex source grammar

The following diagram summarizes the default transformation paths. If there are two paths connecting a pair of suffixes, the longer one is used only if the intermediate file exists or is named in the description.



If the file *x.o* were needed and there were an *x.c* in the description or directory, it would be compiled. If there were also an *x.l*, that grammar would be run through Lex before compiling the result. However, if there were no *x.c* but there were an *x.l*, *make* would discard the intermediate C-language file and use the direct link in the graph above.

It is possible to change the names of some of the compilers used in the default, or the flag arguments with which they are invoked by knowing the macro names used. The compiler names are the macros AS, CC, RC, EC, YACC, YACCR, YACCE, and LEX. The command

```
make CC=newcc
```

will cause the ‘‘newcc’’ command to be used instead of the usual C compiler. The macros CFLAGS, RFLAGS, EFLAGS, YFLAGS, and LFLAGS may be set to cause these commands to be issued with optional flags. Thus,

```
make "CFLAGS=−O"
```

causes the optimizing C compiler to be used.

Example

As an example of the use of *make,* we will present the description file used to maintain the *make* command itself. The code for *make* is spread over a number of C source files and a Yacc grammar. The description file contains:

```
# Description file for the Make command

P = und −3 | opr −r2        # send to GCOS to be printed
FILES = Makefile version.c defs main.c doname.c misc.c files.c dosys.cgram.y lex.c gcos.c
OBJECTS = version.o main.o doname.o misc.o files.o dosys.o gram.o
LIBES= −lS
LINT = lint −p
CFLAGS = −O

make:  $(OBJECTS)
        cc $(CFLAGS) $(OBJECTS) $(LIBES) −o make
        size make

$(OBJECTS):  defs
gram.o: lex.c

cleanup:
        -rm *.o gram.c
        -du

install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make

print:  $(FILES)            # print recently changed files
        pr $? | $P
        touch print

test:
        make −dp | grep −v TIME >1zap
        /usr/bin/make −dp | grep −v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

lint :  dosys.c doname.c files.c main.c misc.c version.c gram.c
        $(LINT) dosys.c doname.c files.c main.c misc.c version.c gram.c
        rm gram.c

arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

Make usually prints out each command before issuing it. The following output results from typing the simple command

```
make
```

in a directory containing only the source and description file:

```
cc  −c version.c
cc  −c main.c
cc  −c doname.c
cc  −c misc.c
cc  −c files.c
cc  −c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc  −c gram.c
cc  version.o main.o doname.o misc.o files.o dosys.o gram.o −lS −o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars were mentioned by name in the description file, *make* found them using its suffix rules and issued the needed commands. The string of digits results from the ‘‘size make’’ command; the printing of the command line itself was suppressed by an @ sign. The @ sign on the *size* command in the description file suppressed the printing of the command, so only the sizes are written.

The last few entries in the description file are useful maintenance sequences. The ‘‘print’’ entry prints only the files that have been changed since the last ‘‘make print’’ command. A zero-length file *print* is maintained to keep track of the time of the printing; the $? macro in the command line then picks up only the names of the files changed since *print* was touched. The printed output can be sent to a different printer or to a file by changing the definition of the *P* macro:

```
make print "P = opr −sp"
        or
make print "P=  cat >zap"
```

Suggestions and Warnings

The most common difficulties arise from *make*'s specific meaning of dependency. If file *x.c* has a ‘‘#include "defs"’’ line, then the object file *x.o* depends on *defs*; the source file *x.c* does not. (If *defs* is changed, it is not necessary to do anything to the file *x.c*, while it is necessary to recreate *x.o*.)

To discover what *make* would do, the ‘‘−n’’ option is very useful. The command

```
make −n
```

orders *make* to print out the commands it would issue without actually taking the time to execute them. If a change to a file is absolutely certain to be benign (e.g., adding a new definition to an include file), the ‘‘−t’’ (touch) option can save a lot of time: instead of issuing a large number of superfluous recompilations, *make* updates the modification times on the affected file. Thus, the command

```
make −ts
```

(‘‘touch silently’’) causes the relevant files to appear up to date. Obvious care is necessary, since this mode of operation subverts the intention of *make* and destroys all memory of the previous relationships.

The debugging flag (‘‘−d’’) causes *make* to print out a very detailed description of what it is doing, including the file times. The output is verbose, and recommended only as a last resort.

Acknowledgments

I would like to thank S. C. Johnson for suggesting this approach to program maintenance control. I would like to thank S. C. Johnson and H. Gajewska for being the prime guinea pigs during development of *make*.

References

1. S. C. Johnson, ‘‘Yacc — Yet Another Compiler-Compiler’’, Bell Laboratories Computing Science Technical Report #32, July 1978.
2. M. E. Lesk, ‘‘Lex — A Lexical Analyzer Generator’’, Computing Science Technical Report #39, October 1975.

Appendix. Suffixes and Transformation Rules

The *make* program itself does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the ‘‘−r’’ flag is used, this table is not used.

The list of suffixes is actually the dependency list for the name ‘‘.SUFFIXES’’; *make* looks for a file with any of the suffixes on the list. If such a file exists, and if there is a transformation rule for that combination, *make* acts as described earlier. The transformation rule names are the concatenation of the two suffixes. The name of the rule to transform a ‘‘*.r*’’ file to a ‘‘*.o*’’ file is thus ‘‘*.r.o*’’. If the rule is present and no explicit command sequence has been given in the user’s description files, the command sequence for the rule ‘‘.r.o’’ is used. If a command is generated by using one of these suffixing rules, the macro $∗ is given the value of the stem (everything but the suffix) of the name of the file to be made, and the macro $< is the name of the dependent that caused the action.

The order of the suffix list is significant, since it is scanned from left to right, and the first name that is formed that has both a file and a rule associated with it is used. If new names are to be appended, the user can just add an entry for ‘‘.SUFFIXES’’ in his own description file; the dependents will be added to the usual list. A ‘‘.SUFFIXES’’ line without any dependents deletes the current list. (It is necessary to clear the current list if the order of names is to be changed).

The following is an excerpt from the default rules file:

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC=yacc
YACCR=yacc −r
YACCE=yacc −e
YFLAGS=
LEX=lex
LFLAGS=
CC=cc
AS=as −
CFLAGS=
RC=ec
RFLAGS=
EC=ec
EFLAGS=
FFLAGS=
.c.o :
	$(CC) $(CFLAGS) −c $<
.e.o .r.o .f.o :
	$(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) −c $<
.s.o :
	$(AS) −o $@ $<
.y.o :
	$(YACC) $(YFLAGS) $<
	$(CC) $(CFLAGS) −c y.tab.c
	rm y.tab.c
	mv y.tab.o $@
.y.c :
	$(YACC) $(YFLAGS) $<
	mv y.tab.c $@
```

UNIX Programming — Second Edition

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper is an introduction to programming on the UNIX† system. The emphasis is on how to write programs that interface to the operating system, either directly or through the standard I/O library. The topics discussed include

- handling command arguments
- rudimentary I/O; the standard input and output
- the standard I/O library; file system access
- low-level I/O: open, read, write, close, seek
- processes: exec, fork, pipes
- signals — interrupts, etc.

There is also an appendix which describes the standard I/O library in detail.

December 3, 1998

†UNIX is a Trademark of Bell Laboratories.

UNIX Programming — Second Edition

Brian W. Kernighan

Dennis M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

This paper describes how to write programs that interface with the UNIX operating system in a non-trivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that attempt to catch interrupts and other signals during execution.

The document collects material which is scattered throughout several sections of *The UNIX Programmer's Manual* [1] for Version 7 UNIX. There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read the language roughly up to the level of *The C Programming Language* [2]. Some of the material in sections 2 through 4 is based on topics covered more carefully there. You should also be familiar with UNIX itself at least to the level of *UNIX for Beginners* [3].

2. BASICS

2.1. Program Arguments

When a C program is run as a command, the arguments on the command line are made available to the function `main` as an argument count `argc` and an array `argv` of pointers to character strings that contain the arguments. By convention, `argv[0]` is the command name itself, so `argc` is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal. (This is essentially the `echo` command.)

```
main(argc, argv)  /* echo arguments */
int argc;
char *argv[];
{
   int i;

   for (i = 1; i < argc; i++)
      printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

`argv` is a pointer to an array whose individual elements are pointers to arrays of characters; each is terminated by `\0`, so they can be treated as strings. The program starts by printing `argv[1]` and loops until it has printed them all.

The argument count and the arguments are parameters to `main`. If you want to keep them around so other routines can get at them, you must copy them to external variables.

2.2. The ''Standard Input'' and ''Standard Output''

The simplest input mechanism is to read the ''standard input,'' which is generally the user's terminal. The function `getchar` returns the next input character each time it is called. A file may be substituted for the terminal by using the < convention: if `prog` uses `getchar`, then the command line

```
prog <file
```

causes `prog` to read `file` instead of the terminal. `prog` itself need know nothing about where its input is coming from. This is also true if the input comes from another program via the pipe mechanism:

```
otherprog | prog
```

provides the standard input for `prog` from the standard output of `otherprog.`

`getchar` returns the value `EOF` when it encounters the end of file (or an error) on whatever you are reading. The value of `EOF` is normally defined to be `-1`, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

Similarly, `putchar(c)` puts the character `c` on the ''standard output,'' which is also by default the terminal. The output can be captured on a file by using >: if `prog` uses `putchar`,

```
prog >outfile
```

writes the standard output on `outfile` instead of the terminal. `outfile` is created if it doesn't exist; if it already exists, its previous contents are overwritten. And a pipe can be used:

```
prog | otherprog
```

puts the standard output of `prog` into the standard input of `otherprog.`

The function `printf`, which formats output in various ways, uses the same mechanism as `putchar` does, so calls to `printf` and `putchar` may be intermixed in any order; the output will appear in the order of the calls.

Similarly, the function `scanf` provides for formatted input conversion; it will read the standard input and break it up into strings, numbers, etc., as desired. `scanf` uses the same mechanism as `getchar`, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with `getchar`, `putchar`, `scanf`, and `printf` may be entirely adequate, and it is almost always enough to get started. This is particularly true if the UNIX pipe facility is used to connect the output of one program to the input of the next. For example, the following program strips out all ascii control characters from its input (except for newline and tab).

```
#include <stdio.h>

main()   /* ccstrip: strip non-graphic characters */
{
   int c;
   while ((c = getchar()) != EOF)
      if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
         putchar(c);
   exit(0);
}
```

The line

```
#include <stdio.h>
```

should appear at the beginning of each source file. It causes the C compiler to read a file (*/usr/include/stdio.h*) of standard routines and symbols that includes the definition of `EOF`.

If it is necessary to treat multiple files, you can use `cat` to collect the files for you:

```
cat file1 file2 ... | ccstrip >output
```

and thus avoid learning how to access files from a program. By the way, the call to `exit` at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Section 6 discusses status returns in more detail.

3. THE STANDARD I/O LIBRARY

The ''Standard I/O Library'' is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

In this section, we will discuss the basics of the standard I/O library. The appendix contains a more complete description of its capabilities.

3.1. File Access

The programs written so far have all read the standard input and written the standard output, which we have assumed are magically pre-defined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is *wc*, which counts the lines, words and characters in a set of files. For instance, the command

```
wc x.c y.c
```

prints the number of lines, words and characters in `x.c` and `y.c` and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the file system names to the I/O statements which actually read the data.

The rules are simple. Before it can be read or written a file has to be *opened* by the standard library function `fopen`. `fopen` takes an external name (like `x.c` or `y.c`), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including `stdio.h` is a structure definition called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE  *fp, *fopen();
```

This says that `fp` is a pointer to a `FILE`, and `fopen` returns a pointer to a `FILE`. (`FILE` is a type name, like `int`, not a structure tag.

The actual call to `fopen` in a program is

```
fp = fopen(name, mode);
```

The first argument of `fopen` is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The only allowable modes are read (`"r"`), write (`"w"`), or append (`"a"`).

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing causes the old contents to be discarded. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file when you don't have permission). If there is any error, `fopen` will return the null pointer value `NULL` (which is defined as zero in `stdio.h`).

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc` and `putc` are the simplest. `getc` returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in `c` the next character from the file referred to by `fp`; it returns `EOF` when it reaches end of file. `putc` is the inverse of `getc`:

```
putc(c, fp)
```

puts the character `c` on the file `fp` and returns `c`. `getc` and `putc` return `EOF` on error.

When a program is started, three files are opened automatically, and file pointers are provided for them. These files are the standard input, the standard output, and the standard error output; the corresponding file pointers are called `stdin`, `stdout`, and `stderr`. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 2.2. `stdin`, `stdout` and `stderr` are pre-defined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type `FILE *` can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write *wc*. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used stand-alone or as part of a larger process.

```
#include <stdio.h>

main(argc, argv)  /* wc: count lines, words, chars */
int argc;
char *argv[];
{
   int c, i, inword;
   FILE *fp, *fopen();
   long linect, wordct, charct;
   long tlinect = 0, twordct = 0, tcharct = 0;

   i = 1;
   fp = stdin;
   do {
      if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
         fprintf(stderr, "wc: can't open %s\n", argv[i]);
         continue;
      }
      linect = wordct = charct = inword = 0;
      while ((c = getc(fp)) != EOF) {
         charct++;
         if (c == '\n')
            linect++;
         if (c == ' ' || c == '\t' || c == '\n')
            inword = 0;
         else if (inword == 0) {
            inword = 1;
            wordct++;
         }
      }
      printf("%7ld %7ld %7ld", linect, wordct, charct);
      printf(argc > 1 ? " %s\n" : "\n", argv[i]);
      fclose(fp);
      tlinect += linect;
      twordct += wordct;
      tcharct += charct;
   } while (++i < argc);
   if (argc > 2)
      printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
   exit(0);
}
```

The function `fprintf` is identical to `printf`, save that the first argument is a file pointer that specifies the file to be written.

The function `fclose` is the inverse of `fopen`; it breaks the connection between the file pointer and the external name that was established by `fopen`, freeing the file pointer for another file. Since there is a limit on the number of files that a program may have open simultaneously, it's a good idea to free things when they are no longer needed. There is also another reason to call `fclose` on an output file — it flushes the buffer in which `putc` is collecting output. (`fclose` is called automatically for each open file when a program terminates normally.)

3.2. Error Handling — Stderr and Exit

`stderr` is assigned to a program in the same way that `stdin` and `stdout` are. Output written on `stderr` appears on the user's terminal even if the standard output is redirected. *wc* writes its diagnostics on `stderr` instead of `stdout` so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The program actually signals errors in another way, using the function `exit` to terminate program execution. The argument of `exit` is available to whatever process called it (see Section 6), so the success or failure of the program can be tested by another program that uses this one as a sub-process. By convention, a return value of 0 signals that all is well; non-zero values signal abnormal situations.

`exit` itself calls `fclose` for each open output file, to flush out any buffered output, then calls a routine named `_exit`. The function `_exit` causes immediate termination without any buffer flushing; it may be called directly if desired.

3.3. Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those we have illustrated above.

Normally output with `putc`, etc., is buffered (except to `stderr`); to force it out immediately, use `fflush(fp)`.

`fscanf` is identical to `scanf`, except that its first argument is a file pointer (as with `fprintf`) that specifies the file from which the input comes; it returns `EOF` at end of file.

The functions `sscanf` and `sprintf` are identical to `fscanf` and `fprintf`, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for `sscanf` and into it for `sprintf`.

`fgets(buf, size, fp)` copies the next line from `fp`, up to and including a newline, into `buf`; at most `size-1` characters are copied; it returns `NULL` at end of file. `fputs(buf, fp)` writes the string in `buf` onto file `fp`.

The function `ungetc(c, fp)` ‘‘pushes back’’ the character `c` onto the input stream `fp`; a subsequent call to `getc`, `fscanf`, etc., will encounter `c`. Only one character of pushback per file is permitted.

4. LOW-LEVEL I/O

This section describes the bottom level of I/O on the UNIX system. The lowest level of I/O in UNIX provides no buffering or any other services; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

4.1. File Descriptors

In the UNIX operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called ‘‘opening’’ the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so (Does the file exist? Do you have permission to access it?), and if all is well, returns a small positive integer called a *file descriptor.* Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. (This is roughly analogous to the use of READ(5,...) and WRITE(6,...) in Fortran.) All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

The file pointers discussed in section 3 are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the ''shell'') runs a program, it opens three files, with file descriptors 0, 1, and 2, called the standard input, the standard output, and the standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without worrying about opening the files.

If I/O is redirected to and from files with < and >, as in

```
prog <infile >outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

4.2. Read and Write

All input and output is done by two functions called `read` and `write`. For both, the first argument is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);

n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than `n` bytes remained to be read. (When the file is a terminal, `read` normally reads only up to the next newline, which is generally less than what was requested.) A return value of zero bytes implies end of file, and `-1` indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time (''unbuffered''), and 512, which corresponds to a physical blocksize on many peripheral devices. This latter size will be most efficient, but even character at a time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define  BUFSIZE  512   /* best size for PDP-11 UNIX */

main()   /* copy input to output */
{
   char  buf[BUFSIZE];
   int   n;

   while ((n = read(0, buf, BUFSIZE)) > 0)
      write(1, buf, n);
   exit(0);
}
```

If the file size is not a multiple of `BUFSIZE`, some `read` will return a smaller number of bytes to be written by `write`; the next call to `read` after that will return zero.

It is instructive to see how `read` and `write` can be used to construct higher level routines like `getchar`, `putchar`, etc. For example, here is a version of `getchar` which does unbuffered input.

```
#define  CMASK 0377  /* for making char's > 0 */

getchar()   /* unbuffered single character input */
{
   char c;

   return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

`c` *must* be declared `char`, because `read` accepts a character pointer. The character being returned must be masked with `0377` to ensure that it is positive; otherwise sign extension may make it negative. (The constant `0377` is appropriate for the PDP-11 but not necessarily for other machines.)

The second version of `getchar` does input in big chunks, and hands out the characters one at a time.

```
#define  CMASK 0377  /* for making char's > 0 */
#define  BUFSIZE  512

getchar()   /* buffered version */
{
   static char buf[BUFSIZE];
   static char *bufp = buf;
   static int  n = 0;

   if (n == 0) {  /* buffer is empty */
      n = read(0, buf, BUFSIZE);
      bufp = buf;
   }
   return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

4.3. Open, Creat, Close, Unlink

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, `open` and `creat` [sic].

`open` is rather like the `fopen` discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an `int`.

```
int fd;

fd = open(name, rwmode);
```

As with `fopen`, the `name` argument is a character string corresponding to the external file name. The access mode argument is different, however: `rwmode` is 0 for read, 1 for write, and 2 for read and write access. `open` returns `-1` if any error occurs; otherwise it returns a valid file descriptor.

It is an error to try to `open` a file that does not exist. The entry point `creat` is provided to create new files, or to re-write old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it was able to create the file called `name`, and `-1` if not. If the file already exists, `creat` will truncate it to zero length; it is not an error to `creat` a file that already exists.

If the file is brand new, `creat` creates it with the *protection mode* specified by the `pmode` argument. In the UNIX file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the UNIX utility *cp*, a program which copies one file to another. (The main simplification is that our version copies only one file, and does not permit the second argument to be a directory.)

```
#define NULL 0
#define BUFSIZE 512
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)  /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
   int   f1, f2, n;
   char  buf[BUFSIZE];

   if (argc != 3)
      error("Usage: cp from to", NULL);
   if ((f1 = open(argv[1], 0)) == -1)
      error("cp: can't open %s", argv[1]);
   if ((f2 = creat(argv[2], PMODE)) == -1)
      error("cp: can't create %s", argv[2]);

   while ((n = read(f1, buf, BUFSIZE)) > 0)
      if (write(f2, buf, n) != n)
         error("cp: write error", NULL);
   exit(0);
}

error(s1, s2)  /* print error message and die */
char *s1, *s2;
{
   printf(s1, s2);
   printf("\n");
   exit(1);
}
```

As we said earlier, there is a limit (typically 15-25) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to re-use file descriptors. The routine `close` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Termination of a program via `exit` or return from the main program closes all open files.

The function `unlink(filename)` removes the file `filename` from the file system.

4.4. Random Access — Seek and Lseek

File I/O is normally sequential: each `read` or `write` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek` provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is `fd` to move to position `offset`, which is taken relative to the location specified by `origin`. Subsequent reading or writing will begin at that position. `offset` is a `long`; `fd` and `origin` are `int`'s. `origin` can be 0, 1, or 2 to specify that `offset` is to be measured from the beginning, from the current position, or from the end of the file respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning (‘‘rewind’’),

```
lseek(fd, 0L, 0);
```

Notice the `0L` argument; it could also be written as `(long) 0`.

With `lseek`, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
   lseek(fd, pos, 0);   /* get to pos */
   return(read(fd, buf, n));
}
```

In pre-version 7 UNIX, the basic entry point to the I/O system is called `seek`. `seek` is identical to `lseek`, except that its `offset` argument is an `int` rather than a `long`. Accordingly, since PDP-11 integers have only 16 bits, the `offset` specified for `seek` is limited to 65,535; for this reason, `origin` values of 3, 4, 5 cause `seek` to multiply the given offset by 512 (the number of bytes in one physical block) and then interpret `origin` as if it were 0, 1, or 2 respectively. Thus to get to an arbitrary place in a large file requires two seeks, first one which selects the block, then one which has `origin` equal to 1 and moves to the desired byte within the block.

4.5. Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of −1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external cell `errno`. The meanings of the various error numbers are listed in the introduction to Section II of the *UNIX Programmer's Manual,* so your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to print out the reason for failure. The routine `perror` will print a message associated with the value of `errno`; more generally, `sys—errno` is an array of character strings which can be indexed by `errno` and printed by your program.

5. PROCESSES

It is often easier to use a program written by someone else than to invent one's own. This section describes how to execute a program from within another.

5.1. The ‘‘System’’ Function

The easiest way to execute a program from another is to use the standard library routine `system`. `system` takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it. For instance, to time-stamp the output of a program,

```
main()
{
    system("date");
    /* rest of processing */
}
```

If the command string has to be built from pieces, the in-memory formatting capabilities of `sprintf` may be useful.

Remember than `getc` and `putc` normally buffer their input; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use `fflush`; for input, see `setbuf` in the appendix.

5.2. Low-Level Process Creation — Execl and Execv

If you’re not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library’s `system` routine is based on.

The most basic operation is to execute another program *without returning*, by using the routine `execl`. To print the date as the last action of a running program, use

```
execl("/bin/date", "date", NULL);
```

The first argument to `execl` is the *file name* of the command; you have to know where it is found in the file system. The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a place-holder. If the command takes arguments, they are strung out after this; the end of the list is marked by a `NULL` argument.

The `execl` call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More realistically, a program might fall into two or more phases that communicate only through temporary files. Here it is natural to make the second pass simply an `execl` call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error, for example if the file can’t be found or is not executable. If you don’t know where `date` is located, say

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of `execl` called `execv` is useful when you don’t know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where `argp` is an array of pointers to the arguments; the last pointer in the array must be `NULL` so `execv` can tell where the list ends. As with `execl`, `filename` is the file in which the program is found, and `argp[0]` is the name of the program. (This arrangement is identical to the `argv` array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like <, >, `*`, `?`, and `[]` in the argument list. If you want these, use `execl` to invoke the shell `sh`, which then does all the work. Construct a string `commandline` that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, /bin/sh. Its argument −c says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in commandline.

5.3. Control of Processes — Fork and Wait

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with execl or execv. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called fork:

```
proc−id = fork();
```

splits the program into two copies, both of which continue to run. The only difference between the two is the value of proc−id, the ‘‘process id.’’ In one of these processes (the ‘‘child’’), proc−id is zero. In the other (the ‘‘parent’’), proc−id is non-zero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork() == 0)
   execl("/bin/sh", "sh", "-c", cmd, NULL);/* in child */
```

And in fact, except for handling errors, this is sufficient. The fork makes two copies of the program. In the child, the value returned by fork is zero, so it calls execl which does the command and then dies. In the parent, fork returns non-zero so it skips the execl. (If there is any error, fork returns −1).

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function wait:

```
int status;

if (fork() == 0)
   execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the execl or fork, or the possibility that there might be more than one child running simultaneously. (The wait returns the process id of the terminated child, if you want to check it against the value returned by fork.) Finally, this fragment doesn't deal with any funny behavior on the part of the child (which is reported in status). Still, these three lines are the heart of the standard library's system routine, which we'll show in a moment.

The status returned by wait encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and non-zero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to exit which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up pointing at the right files, and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither fork nor the exec calls affects open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the execl. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

5.4. Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other reads. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
ls | pr
```

which connects the standard output of `ls` to the standard input of `pr`. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we will illustrate how the pipe connection is established and used.

The system call `pipe` creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int   fd[2];

stat = pipe(fd);
if (stat == -1)
   /* there was an error ... */
```

`fd` is an array of two file descriptors, where `fd[0]` is the read side of the pipe and `fd[1]` is for writing. These may be used in `read`, `write` and `close` calls just like any other file descriptors.

If a process reads a pipe which is empty, it will wait until data arrives; if a process writes into a pipe which is too full, it will wait until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent `read` will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called `popen(cmd, mode)`, which creates a process `cmd` (just as `system` does), and returns a file descriptor that will either read or write that process, according to `mode`. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the `pr` command; subsequent `write` calls using the file descriptor `fout` will send their data to that process through the pipe.

`popen` first creates the the pipe with a `pipe` system call; it then `forks` to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via `execl`) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define  READ  0
#define  WRITE 1
#define  tst(a, b)   (mode == READ ? (b) : (a))
static   int   popen—pid;

popen(cmd, mode)
char  *cmd;
int   mode;
{
   int p[2];

   if (pipe(p) < 0)
      return(NULL);
   if ((popen—pid = fork()) == 0) {
      close(tst(p[WRITE], p[READ]));
      close(tst(0, 1));
      dup(tst(p[READ], p[WRITE]));
      close(tst(p[READ], p[WRITE]));
      execl("/bin/sh", "sh", "-c", cmd, 0);
      —exit(1);   /* disaster has occurred if we get here */
   }
   if (popen—pid == -1)
      return(NULL);
   close(tst(p[READ], p[WRITE]));
   return(tst(p[WRITE], p[READ]));
}
```

The sequence of `closes` in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first `close` closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The `close` closes file descriptor 0, that is, the standard input. `dup` is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the `dup` is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input. (Yes, this is a bit tricky, but it's a standard idiom.) Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write from the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function `pclose` to close the pipe created by `popen`. The main reason for using a separate function rather than `close` is that it is desirable to wait for the termination of the child process. First, the return value from `pclose` indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the `wait` lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd)  /* close pipe fd */
int fd;
{
   register r, (*hstat)(), (*istat)(), (*qstat)();
   int    status;
   extern int popen—pid;

   close(fd);
   istat = signal(SIGINT, SIG—IGN);
   qstat = signal(SIGQUIT, SIG—IGN);
   hstat = signal(SIGHUP, SIG—IGN);
   while ((r = wait(&status)) != popen—pid && r != −1);
   if (r == −1)
      status = −1;
   signal(SIGINT, istat);
   signal(SIGQUIT, qstat);
   signal(SIGHUP, hstat);
   return(status);
}
```

The calls to `signal` make sure that no interrupts, etc., interfere with the waiting process; this is the topic of the next section.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable `popen—pid`; it really should be an array indexed by file descriptor. A `popen` function, with slightly different arguments and return value is available as part of the standard I/O library discussed below. As currently written, it shares the same limitation.

6. SIGNALS — INTERRUPTS AND ALL THAT

This section is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within C about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside-world signals: *interrupt*, which is sent when the DEL character is typed; *quit*, generated by the FS character; *hangup*, caused by hanging up the phone; and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started from the corresponding terminal; unless other arrangements have been made, the signal terminates the process. In the *quit* case, a core image file is written for debugging purposes.

The routine which alters the default action is called `signal`. It has two arguments: the first specifies the signal, and the second specifies how to treat it. The first argument is just a number code, but the second is the address is either a function, or a somewhat strange code that requests that the signal either be ignored, or that it be given the default action. The include file `signal.h` gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
 ...
signal(SIGINT, SIG—IGN);
```

causes interrupts to be ignored, while

```
signal(SIGINT, SIG—DFL);
```

restores the default action of process termination. In all cases, `signal` returns the previous value of the signal. The second argument to `signal` may instead be the name of a function (which has to be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used to allow the program to clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main()
{
   int onintr();

   if (signal(SIGINT, SIG—IGN) != SIG—IGN)
      signal(SIGINT, onintr);

   /* Process ... */

   exit(0);
}

onintr()
{
   unlink(tempfile);
   exit(1);
}
```

Why the test and the double call to `signal`? Recall that signals like interrupt are sent to *all* processes started from a particular terminal. Accordingly, when a program is to be run non-interactively (started by `&`), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the `onintr` routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that `signal` returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command-processing loop. Think of a text editor: interrupting a long printout should not cause it to terminate and lose the work already done. The outline of the code for this case is probably best written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp—buf  sjbuf;

main()
{
   int (*istat)(), onintr();

   istat = signal(SIGINT, SIG—IGN); /* save original status */
   setjmp(sjbuf); /* save current stack position */
   if (istat != SIG—IGN)
      signal(SIGINT, onintr);

   /* main processing loop */
}

onintr()
{
   printf("\nInterrupt\n");
   longjmp(sjbuf);   /* return to saved state */
}
```

The include file `setjmp.h` declares the type `jmp—buf` an object in which the state can be saved. `sjbuf` is such an object; it is an array of some sort. The `setjmp` routine then saves the state of things. When an interrupt occurs, a call is forced to the `onintr` routine, which can print a message, set flags, or whatever. `longjmp` takes as argument an object stored into by `setjmp`, and restores control to the location after the call to `setjmp`, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs. This is necessary; most signals are automatically reset to their default action when they occur.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called on occurrence of a signal sets a flag and then returns instead of calling `exit` or `longjmp`, execution will continue at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the terminal when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that ''execution resumes at the exact point it was interrupted,'' the program would continue reading the terminal until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the terminal read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for ''errors'' which are caused by interrupted system calls. (The ones to watch out for are reads from a terminal, `wait`, and `pause`.) A program whose `onintr` program just sets `intflag`, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar() == EOF)
   if (intflag)
      /* EOF caused by interrupt */
   else
      /* true end-of-file */
```

A final subtlety to keep in mind becomes important when signal-catching is combined with execution of other programs. Suppose a program catches interrupts, and also includes a method (like ''!'' in the editor) whereby other programs can be executed. Then the code should look something like this:

```
if (fork() == 0)
   execl(...);
signal(SIGINT, SIG—IGN);   /* ignore interrupts */
wait(&status); /* until the child is done */
signal(SIGINT, onintr); /* restore interrupts */
```

Why is this? Again, it's not obvious but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read your terminal. But the calling program will also pop out of its wait for the subprogram and read your terminal. Having two processes reading your terminal is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function `system`:

```
#include <signal.h>

system(s)   /* run command string s */
char *s;
{
   int status, pid, w;
   register int (*istat)(), (*qstat)();

   if ((pid = fork()) == 0) {
      execl("/bin/sh", "sh", "-c", s, 0);
      —exit(127);
   }
   istat = signal(SIGINT, SIG—IGN);
   qstat = signal(SIGQUIT, SIG—IGN);
   while ((w = wait(&status)) != pid && w != -1)
      ;
   if (w == -1)
      status = -1;
   signal(SIGINT, istat);
   signal(SIGQUIT, qstat);
   return(status);
}
```

As an aside on declarations, the function `signal` obviously has a rather strange second argument. It is in fact a pointer to a function delivering an integer, and this is also the type of the signal routine itself. The two values `SIG—IGN` and `SIG—DFL` have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the PDP-11; the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

```
#define  SIG—DFL  (int (*)())0
#define  SIG—IGN  (int (*)())1
```

References

[1] K. L. Thompson and D. M. Ritchie, *The UNIX Programmer's Manual,* Bell Laboratories, 1978.

[2] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, Inc., 1978.

[3] B. W. Kernighan, ''UNIX for Beginners — Second Edition.'' Bell Laboratories, 1978.

Appendix — The Standard I/O Library

D. M. Ritchie

Bell Laboratories
Murray Hill, New Jersey 07974

The standard I/O library was designed with the following goals in mind.

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to machines other than the PDP-11 running a version of UNIX.

1. General Usage

Each program using the library must have the line

```
#include <stdio.h>
```

which defines certain macros and variables. The routines are in the normal C library, so no special library argument is needed for loading. All names in the include file intended only for internal use begin with an underscore _ to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

`stdin` The name of the standard input file

`stdout` The name of the standard output file

`stderr` The name of the standard error file

`EOF` is actually −1, and is the value returned by the read routines on end-of-file or error.

`NULL` is a notation for the null pointer, returned by pointer-valued functions to indicate an error

`FILE` expands to `struct _iob` and is a useful shorthand when declaring pointers to streams.

`BUFSIZ` is a number (viz. 512) of the size suitable for an I/O buffer supplied by the user. See `setbuf`, below.

`getc, getchar, putc, putchar, feof, ferror, fileno`
are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names `stdin`, `stdout`, and `stderr` are in effect constants and may not be assigned to.

2. Calls

`FILE *fopen(filename, type) char *filename, *type;`
opens the file and, if needed, allocates a buffer for it. `filename` is a character string specifying the name. `type` is a character string (not a single character). It may be `"r"`, `"w"`, or `"a"` to indicate intent to read, write, or append. The value returned is a file pointer. If it is `NULL` the attempt to open failed.

`FILE *freopen(filename, type, ioptr) char *filename, *type; FILE *ioptr;`
The stream named by `ioptr` is closed, if necessary, and then reopened as if by `fopen`. If the attempt to open fails, `NULL` is returned, otherwise `ioptr`, which will now refer to the new file. Often the reopened stream is `stdin` or `stdout`.

`int getc(ioptr) FILE *ioptr;`
returns the next character from the stream named by `ioptr`, which is a pointer to a file such as returned by `fopen`, or the name `stdin`. The integer `EOF` is returned on end-of-file or when an error occurs. The null character `\0` is a legal character.

`int fgetc(ioptr) FILE *ioptr;`
acts like `getc` but is a genuine function, not a macro, so it can be pointed to, passed as an argument, etc.

`putc(c, ioptr) FILE *ioptr;`
`putc` writes the character `c` on the output stream named by `ioptr`, which is a value returned from `fopen` or perhaps `stdout` or `stderr`. The character is returned as value, but `EOF` is returned on error.

`fputc(c, ioptr) FILE *ioptr;`
acts like `putc` but is a genuine function, not a macro.

`fclose(ioptr) FILE *ioptr;`
The file corresponding to `ioptr` is closed after any buffers are emptied. A buffer allocated by the I/O system is freed. `fclose` is automatic on normal termination of the program.

`fflush(ioptr) FILE *ioptr;`
Any buffered information on the (output) stream named by `ioptr` is written out. Output files are normally buffered if and only if they are not directed to the terminal; however, `stderr` always starts off unbuffered and remains so unless `setbuf` is used, or unless it is reopened.

`exit(errcode);`
terminates the process and returns its argument as status to the parent. This is a special version of the routine which calls `fflush` for each output file. To terminate without flushing, use `_exit`.

`feof(ioptr) FILE *ioptr;`
returns non-zero when end-of-file has occurred on the specified input stream.

`ferror(ioptr) FILE *ioptr;`
returns non-zero when an error has occurred while reading or writing the named stream. The error indication lasts until the file has been closed.

`getchar();`
is identical to `getc(stdin)`.

`putchar(c);`
is identical to `putc(c, stdout)`.

`char *fgets(s, n, ioptr) char *s; FILE *ioptr;`
reads up to `n-1` characters from the stream `ioptr` into the character pointer `s`. The read terminates with a newline character. The newline character is placed in the buffer followed by a null character. `fgets` returns the first argument, or `NULL` if error or end-of-file occurred.

`fputs(s, ioptr) char *s; FILE *ioptr;`
writes the null-terminated string (character array) `s` on the stream `ioptr`. No newline is appended. No value is returned.

`ungetc(c, ioptr) FILE *ioptr;`
The argument character `c` is pushed back on the input stream named by `ioptr`. Only one character may be pushed back.

`printf(format, a1, ...) char *format;`
`fprintf(ioptr, format, a1, ...) FILE *ioptr; char *format;`
`sprintf(s, format, a1, ...)char *s, *format;`
`printf` writes on the standard output. `fprintf` writes on the named output stream. `sprintf` puts characters in the character array (string) named by `s`. The specifications are as described in section `printf`(3) of the *UNIX Programmer's Manual.*

`scanf(format, a1, ...) char *format;`
`fscanf(ioptr, format, a1, ...) FILE *ioptr; char *format;`
`sscanf(s, format, a1, ...) char *s, *format;`

`scanf` reads from the standard input. `fscanf` reads from the named input stream. `sscanf` reads from the character string supplied as `s`. `scanf` reads characters, interprets them according to a format, and stores the results in its arguments. Each routine expects as arguments a control string `format`, and a set of arguments, *each of which must be a pointer,* indicating where the converted input should be stored.

`scanf` returns as its value the number of successfully matched and assigned input items. This can be used to decide how many input items were found. On end of file, `EOF` is returned; note that this is different from 0, which means that the next input character does not match what was called for in the control string.

`fread(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;`

reads `nitems` of data beginning at `ptr` from file `ioptr`. No advance notification that binary I/O is being done is required; when, for portability reasons, it becomes required, it will be done by adding an additional character to the mode-string on the `fopen` call.

`fwrite(ptr, sizeof(*ptr), nitems, ioptr) FILE *ioptr;`

Like `fread`, but in the other direction.

`rewind(ioptr) FILE *ioptr;`

rewinds the stream named by `ioptr`. It is not very useful except on input, since a rewound output file is still open only for output.

`system(string) char *string;`

The `string` is executed by the shell as if typed at the terminal.

`getw(ioptr) FILE *ioptr;`

returns the next word from the input stream named by `ioptr`. `EOF` is returned on end-of-file or error, but since this a perfectly good integer `feof` and `ferror` should be used. A ''word'' is 16 bits on the PDP-11.

`putw(w, ioptr) FILE *ioptr;`

writes the integer `w` on the named output stream.

`setbuf(ioptr, buf) FILE *ioptr; char *buf;`

`setbuf` may be used after a stream has been opened but before I/O has started. If `buf` is `NULL`, the stream will be unbuffered. Otherwise the buffer supplied will be used. It must be a character array of sufficient size:

```
char  buf[BUFSIZ];
```

`fileno(ioptr) FILE *ioptr;`

returns the integer file descriptor associated with the file.

`fseek(ioptr, offset, ptrname) FILE *ioptr; long offset;`

The location of the next byte in the stream named by `ioptr` is adjusted. `offset` is a long integer. If `ptrname` is 0, the offset is measured from the beginning of the file; if `ptrname` is 1, the offset is measured from the current read or write pointer; if `ptrname` is 2, the offset is measured from the end of the file. The routine accounts properly for any buffering. (When this routine is used on non-UNIX systems, the offset must be a value returned from `ftell` and the ptrname must be 0).

`long ftell(ioptr) FILE *ioptr;`

The byte offset, measured from the beginning of the file, associated with the named stream is returned. Any buffering is properly accounted for. (On non-UNIX systems the value of this call is useful only for handing to `fseek`, so as to position the file to the same place it was when `ftell` was called.)

`getpw(uid, buf) char *buf;`
The password file is searched for the given integer user ID. If an appropriate line is found, it is copied into the character array `buf`, and 0 is returned. If no line is found corresponding to the user ID then 1 is returned.

`char *malloc(num);`
allocates `num` bytes. The pointer returned is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available.

`char *calloc(num, size);`
allocates space for `num` items each of size `size`. The space is guaranteed to be set to 0 and the pointer is sufficiently well aligned to be usable for any purpose. `NULL` is returned if no space is available .

`cfree(ptr) char *ptr;`
Space is returned to the pool used by `calloc`. Disorder can be expected if the pointer was not obtained from `calloc`.

The following are macros whose definitions may be obtained by including `<ctype.h>`.

`isalpha(c)` returns non-zero if the argument is alphabetic.

`isupper(c)` returns non-zero if the argument is upper-case alphabetic.

`islower(c)` returns non-zero if the argument is lower-case alphabetic.

`isdigit(c)` returns non-zero if the argument is a digit.

`isspace(c)` returns non-zero if the argument is a spacing character: tab, newline, carriage return, vertical tab, form feed, space.

`ispunct(c)` returns non-zero if the argument is any punctuation character, i.e., not a space, letter, digit or control character.

`isalnum(c)` returns non-zero if the argument is a letter or a digit.

`isprint(c)` returns non-zero if the argument is printable — a letter, digit, or punctuation character.

`iscntrl(c)` returns non-zero if the argument is a control character.

`isascii(c)` returns non-zero if the argument is an ascii character, i.e., less than octal 0200.

`toupper(c)` returns the upper-case character corresponding to the lower-case letter `c`.

`tolower(c)` returns the lower-case character corresponding to the upper-case letter `c`.

A Tutorial Introduction to ADB

J. F. Maranzano

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Debugging tools generally provide a wealth of information about the inner workings of programs. These tools have been available on UNIX† to allow users to examine ‘‘core’’ files that result from aborted programs. A new debugging program, ADB, provides enhanced capabilities to examine "core" and other program files in a variety of formats, run programs with embedded breakpoints and patch files.

ADB is an indispensable but complex tool for debugging crashed systems and/or programs. This document provides an introduction to ADB with examples of its use. It explains the various formatting options, techniques for debugging C programs, examples of printing file system information and patching.

May 5, 1977

†UNIX is a Trademark of Bell Laboratories.

A Tutorial Introduction to ADB

J. F. Maranzano

S. R. Bourne

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

ADB is a new debugging program that is available on UNIX. It provides capabilities to look at ''core'' files resulting from aborted programs, print output in a variety of formats, patch files, and run programs with embedded breakpoints. This document provides examples of the more useful features of ADB. The reader is expected to be familiar with the basic commands on UNIX† with the C language, and with References 1, 2 and 3.

2. A Quick Survey

2.1. Invocation

ADB is invoked as:

adb objfile corefile

where *objfile* is an executable UNIX file and *corefile* is a core image file. Many times this will look like:

adb a.out core

or more simply:

adb

where the defaults are *a.out* and *core* respectively. The filename minus (–) means ignore this argument as in:

adb – core

ADB has requests for examining locations in either file. The **?** request examines the contents of *objfile,* the **/** request examines the *corefile.* The general form of these requests is:

address ? format

or

address / format

2.2. Current Address

ADB maintains a current address, called dot, similar in function to the current pointer in the UNIX editor. When an address is entered, the current address is set to that location, so that:

0126?i

sets dot to octal 126 and prints the instruction at that address. The request:

†UNIX is a Trademark of Bell Laboratories.

.,10/d

prints 10 decimal numbers starting at dot. Dot ends up referring to the address of the last item printed. When used with the **?** or **/** requests, the current address can be advanced by typing newline; it can be decremented by typing ^.

Addresses are represented by expressions. Expressions are made up from decimal, octal, and hexadecimal integers, and symbols from the program under test. These may be combined with the operators +, −, *, % (integer division), & (bitwise and), | (bitwise inclusive or), # (round up to the next multiple), and ~ (not). (All arithmetic within ADB is 32 bits.) When typing a symbolic address for a C program, the user can type *name* or *_name;* ADB will recognize both forms.

2.3. Formats

To print data, a user specifies a collection of letters and characters that describe the format of the printout. Formats are "remembered" in the sense that typing a request without one will cause the new printout to appear in the previous format. The following are the most commonly used format letters.

b	**one byte in octal**
c	**one byte as a character**
o	**one word in octal**
d	**one word in decimal**
f	**two words in floating point**
i	**PDP 11 instruction**
s	**a null terminated character string**
a	**the value of dot**
u	**one word as unsigned integer**
n	**print a newline**
r	**print a blank space**
^	**backup dot**

(Format letters are also available for "long" values, for example, '**D**' for long decimal, and '**F**' for double floating point.) For other formats see the ADB manual.

2.4. General Request Meanings

The general form of a request is:

address,count command modifier

which sets 'dot' to *address* and executes the command *count* times.

The following table illustrates some general ADB command meanings:

Command	**Meaning**
?	**Print contents from *a.out* file**
/	**Print contents from *core* file**
=	**Print value of "dot"**
:	**Breakpoint control**
$	**Miscellaneous requests**
;	**Request separator**
!	**Escape to shell**

ADB catches signals, so a user cannot use a quit signal to exit from ADB. The request $q or $Q (or cntl-D) must be used to exit from ADB.

3. Debugging C Programs

3.1. Debugging A Core Image

Consider the C program in Figure 1. The program is used to illustrate a common error made by C programmers. The object of the program is to change the lower case "t" to upper case in the string pointed to by *charp* and then write the character string to the file indicated by argument 1. The bug shown is that the character "T" is stored in the pointer *charp* instead of the string pointed to by *charp.* Executing the program produces a core file because of an out of bounds memory reference.

ADB is invoked by:

adb a.out core

The first debugging request:

$c

is used to give a C backtrace through the subroutines called. As shown in Figure 2 only one function (*main*) was called and the arguments *argc* and *argv* have octal values 02 and 0177762 respectively. Both of these values look reasonable; 02 = two arguments, 0177762 = address on stack of parameter vector.
The next request:

$C

is used to give a C backtrace plus an interpretation of all the local variables in each function and their values in octal. The value of the variable *cc* looks incorrect since *cc* was declared as a character.

The next request:

$r

prints out the registers including the program counter and an interpretation of the instruction at that location.

The request:

$e

prints out the values of all external variables.

A map exists for each file handled by ADB. The map for the *a.out* file is referenced by **?** whereas the map for *core* file is referenced by **/**. Furthermore, a good rule of thumb is to use **?** for instructions and **/** for data when looking at programs. To print out information about the maps type:

$m

This produces a report of the contents of the maps. More about these maps later.

In our example, it is useful to see the contents of the string pointed to by *charp.* This is done by:

***charp/s**

which says use *charp* as a pointer in the *core* file and print the information as a character string. This printout clearly shows that the character buffer was incorrectly overwritten and helps identify the error. Printing the locations around *charp* shows that the buffer is unchanged but that the pointer is destroyed. Using ADB similarly, we could print information about the arguments to a function. The request:

main.argc/d

prints the decimal *core* image value of the argument *argc* in the function *main.*
The request:

***main.argv,3/o**

prints the octal values of the three consecutive cells pointed to by *argv* in the function *main.* Note that these values are the addresses of the arguments to main. Therefore:

0177770/s

prints the ASCII value of the first argument. Another way to print this value would have been

***"/s**

The " means ditto which remembers the last address typed, in this case *main.argc* ; the ***** instructs ADB to use the address field of the *core* file as a pointer.

The request:

.=o

prints the current address (not its contents) in octal which has been set to the address of the first argument. The current address, dot, is used by ADB to "remember" its current location. It allows the user to reference locations relative to the current address, for example:

.−10/d

3.2. Multiple Functions

Consider the C program illustrated in Figure 3. This program calls functions *f, g,* and *h* until the stack is exhausted and a core image is produced.

Again you can enter the debugger via:

adb

which assumes the names *a.out* and *core* for the executable file and core image file respectively. The request:

$c

will fill a page of backtrace references to *f, g,* and *h.* Figure 4 shows an abbreviated list (typing *DEL* will terminate the output and bring you back to ADB request level).

The request:

,5$C

prints the five most recent activations.

Notice that each function (*f,g,h*) has a counter of the number of times it was called.

The request:

fcnt/d

prints the decimal value of the counter for the function *f.* Similarly *gcnt* and *hcnt* could be printed. To print the value of an automatic variable, for example the decimal value of *x* in the last call of the function *h,* type:

h.x/d

It is currently not possible in the exported version to print stack frames other than the most recent activation of a function. Therefore, a user can print everything with **$C** or the occurrence of a variable in the most recent call of a function. It is possible with the **$C** request, however, to print the stack frame starting at some address as **address$C.**

3.3. Setting Breakpoints

Consider the C program in Figure 5. This program, which changes tabs into blanks, is adapted from *Software Tools* by Kernighan and Plauger, pp. 18-27.

We will run this program under the control of ADB (see Figure 6a) by:

adb a.out –

Breakpoints are set in the program as:

address:b [request]

The requests:

> **settab+4:b**
> **fopen+4:b**
> **getc+4:b**
> **tabpos+4:b**

set breakpoints at the start of these functions. C does not generate statement labels. Therefore it is currently not possible to plant breakpoints at locations other than function entry points without a knowledge of the code generated by the C compiler. The above addresses are entered as **symbol+4** so that they will appear in any C backtrace since the first instruction of each function is a call to the C save routine (*csv*). Note that some of the functions are from the C library.

To print the location of breakpoints one types:

> **$b**

The display indicates a *count* field. A breakpoint is bypassed *count −1* times before causing a stop. The *command* field indicates the ADB requests to be executed each time the breakpoint is encountered. In our example no *command* fields are present.

By displaying the original instructions at the function *settab* we see that the breakpoint is set after the jsr to the C save routine. We can display the instructions using the ADB request:

> **settab,5?ia**

This request displays five instructions starting at *settab* with the addresses of each location displayed. Another variation is:

> **settab,5?i**

which displays the instructions with only the starting address.

Notice that we accessed the addresses from the *a.out* file with the **?** command. In general when asking for a printout of multiple items, ADB will advance the current address the number of bytes necessary to satisfy the request; in the above example five instructions were displayed and the current address was advanced 18 (decimal) bytes.

To run the program one simply types:

> **:r**

To delete a breakpoint, for instance the entry to the function *settab,* one types:

> **settab+4:d**

To continue execution of the program from the breakpoint type:

> **:c**

Once the program has stopped (in this case at the breakpoint for *fopen),* ADB requests can be used to display the contents of memory. For example:

> **$C**

to display a stack trace, or:

> **tabs,3/8o**

to print three lines of 8 locations each from the array called *tabs.* By this time (at location *fopen)* in the C program, *settab* has been called and should have set a one in every eighth location of *tabs.*

3.4. Advanced Breakpoint Usage

We continue execution of the program with:

> **:c**

See Figure 6b. *Getc* is called three times and the contents of the variable *c* in the function *main* are displayed each time. The single character on the left hand edge is the output from the C program. On the third occurrence of *getc* the program stops. We can look at the full buffer of characters by typing:

ibuf+6/20c

When we continue the program with:

:c

we hit our first breakpoint at *tabpos* since there is a tab following the "This" word of the data.

Several breakpoints of *tabpos* will occur until the program has changed the tab into equivalent blanks. Since we feel that *tabpos* is working, we can remove the breakpoint at that location by:

tabpos+4:d

If the program is continued with:

:c

it resumes normal execution after ADB prints the message

a.out:running

The UNIX quit and interrupt signals act on ADB itself rather than on the program being debugged. If such a signal occurs then the program being debugged is stopped and control is returned to ADB. The signal is saved by ADB and is passed on to the test program if:

:c

is typed. This can be useful when testing interrupt handling routines. The signal is not passed on to the test program if:

:c 0

is typed.

Now let us reset the breakpoint at *settab* and display the instructions located there when we reach the breakpoint. This is accomplished by:

settab+4:b settab,5?ia *

It is also possible to execute the ADB requests for each occurrence of the breakpoint but only stop after the third occurrence by typing:

getc+4,3:b main.c?C *

This request will print the local variable *c* in the function *main* at each occurrence of the breakpoint. The semicolon is used to separate multiple ADB requests on a single line.

Warning: setting a breakpoint causes the value of dot to be changed; executing the program under ADB does not change dot. Therefore:

settab+4:b .,5?ia
fopen+4:b

will print the last thing dot was set to (in the example *fopen+4*) *not* the current location (*settab+4*) at which the program is executing.

---

* Owing to a bug in early versions of ADB (including the version distributed in Generic 3 UNIX) these statements must be written as:

settab+4:b settab,5?ia;0
getc+4,3:bmain.c?C;0
settab+4:b settab,5?ia; ptab/o;0

Note that **;0** will set dot to zero and stop at the breakpoint.

A breakpoint can be overwritten without first deleting the old breakpoint. For example:

settab+4:b settab,5?ia; ptab/o *

could be entered after typing the above requests.

Now the display of breakpoints:

$b

shows the above request for the *settab* breakpoint. When the breakpoint at *settab* is encountered the ADB requests are executed. Note that the location at *settab+4* has been changed to plant the breakpoint; all the other locations match their original value.

Using the functions, *f, g* and *h* shown in Figure 3, we can follow the execution of each function by planting non-stopping breakpoints. We call ADB with the executable program of Figure 3 as follows:

adb ex3 –

Suppose we enter the following breakpoints:

```
h+4:b      hcnt/d;  h.hi/;  h.hr/
g+4:b      gcnt/d;  g.gi/;  g.gr/
f+4:b      fcnt/d;  f.fi/;  f.fr/
:r
```

Each request line indicates that the variables are printed in decimal (by the specification **d**). Since the format is not changed, the **d** can be left off all but the first request.

The output in Figure 7 illustrates two points. First, the ADB requests in the breakpoint line are not examined until the program under test is run. That means any errors in those ADB requests is not detected until run time. At the location of the error ADB stops running the program.

The second point is the way ADB handles register variables. ADB uses the symbol table to address variables. Register variables, like *f.fr* above, have pointers to uninitialized places on the stack. Therefore the message "symbol not found".

Another way of getting at the data in this example is to print the variables used in the call as:

```
f+4:b      fcnt/d;  f.a/;  f.b/;  f.fi/
g+4:b      gcnt/d;  g.p/;  g.q/;  g.gi/
:c
```

The operator / was used instead of ? to read values from the *core* file. The output for each function, as shown in Figure 7, has the same format. For the function *f*, for example, it shows the name and value of the *external* variable *fcnt*. It also shows the address on the stack and value of the variables *a, b* and *fi*.

Notice that the addresses on the stack will continue to decrease until no address space is left for program execution at which time (after many pages of output) the program under test aborts. A display with names would be produced by requests like the following:

```
f+4:b      fcnt/d;  f.a/"a="d;  f.b/"b="d;  f.fi/"fi="d
```

In this format the quoted string is printed literally and the **d** produces a decimal display of the variables. The results are shown in Figure 7.

3.5. Other Breakpoint Facilities

- Arguments and change of standard input and output are passed to a program as:

  :r arg1 arg2 ... <infile >outfile

  This request kills any existing program under test and starts the *a.out* afresh.

- The program being debugged can be single stepped by:

  :s

  If necessary, this request will start up the program being debugged and stop after executing the first instruction.

- ADB allows a program to be entered at a specific address by typing:

  address:r

- The count field can be used to skip the first *n* breakpoints as:

  ,n:r

  The request:

  ,n:c

  may also be used for skipping the first *n* breakpoints when continuing a program.

- A program can be continued at an address different from the breakpoint by:

  address:c

- The program being debugged runs as a separate process and can be killed by:

  :k

4. Maps

UNIX supports several executable file formats. These are used to tell the loader how to load the program file. File type 407 is the most common and is generated by a C compiler invocation such as **cc pgm.c**. A 410 file is produced by a C compiler command of the form **cc -n pgm.c**, whereas a 411 file is produced by **cc -i pgm.c**. ADB interprets these different file formats and provides access to the different segments through a set of maps (see Figure 8). To print the maps type:

$m

In 407 files, both text (instructions) and data are intermixed. This makes it impossible for ADB to differentiate data from instructions and some of the printed symbolic addresses look incorrect; for example, printing data addresses as offsets from routines.

In 410 files (shared text), the instructions are separated from data and **?*** accesses the data part of the *a.out* file. The **?*** request tells ADB to use the second part of the map in the *a.out* file. Accessing data in the *core* file shows the data after it was modified by the execution of the program. Notice also that the data segment may have grown during program execution.

In 411 files (separated I & D space), the instructions and data are also separated. However, in this case, since data is mapped through a separate set of segmentation registers, the base of the data segment is also relative to address zero. In this case since the addresses overlap it is necessary to use the **?*** operator to access the data space of the *a.out* file. In both 410 and 411 files the corresponding core file does not contain the program text.

Figure 9 shows the display of three maps for the same program linked as a 407, 410, 411 respectively. The b, e, and f fields are used by ADB to map addresses into file addresses. The "f1" field is the length of the header at the beginning of the file (020 bytes for an *a.out* file and 02000 bytes for a *core* file). The "f2" field is the displacement from the beginning of the file to the data. For a 407 file with mixed text and data this is the same as the length of the header; for 410 and 411 files this is the length of the header plus the size of the text portion.

The "b" and "e" fields are the starting and ending locations for a segment. Given an address, A, the location in the file (either *a.out* or *core*) is calculated as:

b1≤A≤e1 ≥ file address = (A– b1)+f1
b2≤A≤e2 ≥ file address = (A– b2)+f2

A user can access locations by using the ADB defined variables. The **$v** request prints the variables initialized by ADB:

b	**base address of data segment**
d	**length of the data segment**
s	**length of the stack**
t	**length of the text**
m	**execution type (407,410,411)**

In Figure 9 those variables not present are zero. Use can be made of these variables by expressions such as:

<b

in the address field. Similarly the value of the variable can be changed by an assignment request such as:

02000>b

that sets **b** to octal 2000. These variables are useful to know if the file under examination is an executable or *core* image file.

ADB reads the header of the *core* image file to find the values for these variables. If the second file specified does not seem to be a *core* file, or if it is missing then the header of the executable file is used instead.

5. Advanced Usage

It is possible with ADB to combine formatting requests to provide elaborate displays. Below are several examples.

5.1. Formatted dump

The line:

<b,– 1/4o4^8Cn

prints 4 octal words followed by their ASCII interpretation from the data space of the core image file. Broken down, the various request pieces mean:

<b	The base address of the data segment.
<b,– 1	Print from the base address to the end of file. A negative count is used here and elsewhere to loop indefinitely or until some error condition (like end of file) is detected.

The format **4o4^8Cn** is broken down as follows:

4o	Print 4 octal locations.
4^	Backup the current address 4 locations (to the original start of the field).
8C	Print 8 consecutive characters using an escape convention; each character in the range 0 to 037 is printed as @ followed by the corresponding character in the range 0140 to 0177. An @ is printed as @@.
n	Print a newline.

The request:

```
<b,<d/4o4^8Cn
```

could have been used instead to allow the printing to stop at the end of the data segment (<d provides the data segment size in bytes).

The formatting requests can be combined with ADB's ability to read in a script to produce a core image dump script. ADB is invoked as:

```
adb a.out core < dump
```

to read in a script file, *dump,* of requests. An example of such a script is:

```
120$w
4095$s
$v
=3n
$m
=3n"C Stack Backtrace"
$C
=3n"C External Variables"
$e
=3n"Registers"
$r
0$s
=3n"Data Segment"
<b,–1/8ona
```

The request **120$w** sets the width of the output to 120 characters (normally, the width is 80 characters). ADB attempts to print addresses as:

symbol + offset

The request **4095$s** increases the maximum permissible offset to the nearest symbolic address from 255 (default) to 4095. The request **=** can be used to print literal strings. Thus, headings are provided in this *dump* program with requests of the form:

=3n"C Stack Backtrace"

that spaces three lines and prints the literal string. The request **$v** prints all non-zero ADB variables (see Figure 8). The request **0$s** sets the maximum offset for symbol matches to zero thus suppressing the printing of symbolic labels in favor of octal values. Note that this is only done for the printing of the data segment. The request:

```
<b,–1/8ona
```

prints a dump from the base of the data segment to the end of file with an octal address field and eight octal numbers per line.

Figure 11 shows the results of some formatting requests on the C program of Figure 10.

5.2. Directory Dump

As another illustration (Figure 12) consider a set of requests to dump the contents of a directory (which is made up of an integer *inumber* followed by a 14 character name):

```
adb dir –
=n8t"Inum"8t"Name"
0,–1? u8t14cn
```

In this example, the **u** prints the *inumber* as an unsigned decimal integer, the **8t** means that ADB will space to the next multiple of 8 on the output line, and the **14c** prints the 14 character file name.

5.3. Ilist Dump

Similarly the contents of the *ilist* of a file system, (e.g. /dev/src, on UNIX systems distributed by the UNIX Support Group; see UNIX Programmer's Manual Section V) could be dumped with the following set of requests:

adb /dev/src –
02000>b
?m <b
<b,– 1?"flags"8ton"links,uid,gid"8t3bn",size"8tbrdn"addr"8t8un"times"8t2Y2na

In this example the value of the base for the map was changed to 02000 (by saying **?m<b**) since that is the start of an *ilist* within a file system. An artifice (**brd** above) was used to print the 24 bit size field as a byte, a space, and a decimal integer. The last access time and last modify time are printed with the **2Y** operator. Figure 12 shows portions of these requests as applied to a directory and file system.

5.4. Converting values

ADB may be used to convert values from one representation to another. For example:

072 = odx

will print

072 58 #3a

which is the octal, decimal and hexadecimal representations of 072 (octal). The format is remembered so that typing subsequent numbers will print them in the given formats. Character values may be converted similarly, for example:

'a' = co

prints

a 0141

It may also be used to evaluate expressions but be warned that all binary operators have the same precedence which is lower than that for unary operators.

6. Patching

Patching files with ADB is accomplished with the *write,* **w** or **W**, request (which is not like the *ed* editor write command). This is often used in conjunction with the *locate,* **l** or **L** request. In general, the request syntax for **l** and **w** are similar as follows:

?l value

The request **l** is used to match on two bytes, **L** is used for four bytes. The request **w** is used to write two bytes, whereas **W** writes four bytes. The **value** field in either *locate* or *write* requests is an expression. Therefore, decimal and octal numbers, or character strings are supported.

In order to modify a file, ADB must be called as:

adb –w file1 file2

When called with this option, *file1* and *file2* are created if necessary and opened for both reading and writing.

For example, consider the C program shown in Figure 10. We can change the word "This" to "The " in the executable file for this program, *ex7*, by using the following requests:

adb –w ex7 –
?l 'Th'
?W 'The '

The request **?l** starts at dot and stops at the first match of "Th" having set dot to the address of the location found. Note the use of **?** to write to the *a.out* file. The form **?*** would have been used for a 411 file.

More frequently the request will be typed as:

?l 'Th'; ?s

and locates the first occurrence of "Th" and print the entire string. Execution of this ADB request will set dot to the address of the "Th" characters.

As another example of the utility of the patching facility, consider a C program that has an internal logic flag. The flag could be set by the user through ADB and the program run. For example:

```
adb a.out –
:s arg1 arg2
flag/w 1
:c
```

The **:s** request is normally used to single step through a process or start a process in single step mode. In this case it starts *a.out* as a subprocess with arguments **arg1** and **arg2**. If there is a subprocess running ADB writes to it rather than to the file so the **w** request causes *flag* to be changed in the memory of the subprocess.

7. Anomalies

Below is a list of some strange things that users should be aware of.

1. Function calls and arguments are put on the stack by the C save routine. Putting breakpoints at the entry point to routines means that the function appears not to have been called when the breakpoint occurs.
2. When printing addresses, ADB uses either text or data symbols from the *a.out* file. This sometimes causes unexpected symbol names to be printed with data (e.g. *savr5+022*). This does not happen if **?** is used for text (instructions) and **/** for data.
3. ADB cannot handle C register variables in the most recently activated function.

8. Acknowledgements

The authors are grateful for the thoughtful comments on how to organize this document from R. B. Brandt, E. N. Pinson and B. A. Tague. D. M. Ritchie made the system changes necessary to accommodate tracing within ADB. He also participated in discussions during the writing of ADB. His earlier work with DB and CDB led to many of the features found in ADB.

9. References

1. D. M. Ritchie and K. Thompson, ''The UNIX Time-Sharing System,'' CACM, July, 1974.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978.
3. K. Thompson and D. M. Ritchie, UNIX Programmer's Manual - 7th Edition, 1978.
4. B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, 1976.

Figure 1: C program with pointer bug

```
struct buf {
        int fildes;
        int nleft;
        char *nextp;
        char buff[512];
        }bb;
struct buf *obuf;

char *charp "this is a sentence.";

main(argc,argv)
int argc;
char **argv;
{
        char    cc;

        if(argc < 2) {
                printf("Input file missing\n");
                exit(8);
        }

        if((fcreat(argv[1],obuf)) < 0){
                printf("%s : not found\n", argv[1]);
                exit(8);
        }
        charp = ´T´;
printf("debug 1 %s\n",charp);
        while(cc=  *charp++)
                putc(cc,obuf);
        fflush(obuf);
}
```

Figure 2: ADB output for C program of Figure 1

```
adb a.out core
$c
~main(02,0177762)
$C
~main(02,0177762)
        argc:       02
        argv:       0177762
        cc:         02124
$r
ps      0170010
pc      0204      ~main+0152
sp      0177740
r5      0177752
r4      01
r3      0
r2      0
r1      0
r0      0124
~main+0152:     mov     _obuf,(sp)
$e
savr5:     0
_obuf:     0
_charp:    0124
_errno:    0
_fout:     0
$m
text map   `ex1´
b1 = 0              e1    = 02360          f1 = 020
b2 = 0              e2    = 02360          f2 = 020
data map   `core1´
b1 = 0              e1    = 03500          f1 = 02000
b2 = 0175400        e2    = 0200000                f2 = 05500
*charp/s
0124:           TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTLx     Nh@x&_
~
charp/s
_charp:         T

_charp+02:      this is a sentence.

_charp+026:     Input file missing
main.argc/d
0177756: 2
*main.argv/3o
0177762: 0177770  0177776  0177777
0177770/s
0177770: a.out
*main.argv/3o
0177762: 0177770  0177776  0177777
*"/s
0177770: a.out
 .=o
                0177770
 .–10/d
0177756: 2
$q
```

Figure 3: Multiple function C program for stack trace illustration

```
int     fcnt,gcnt,hcnt;
h(x,y)
{
        int hi; register int hr;
        hi = x+1;
        hr = x−y+1;
        hcnt++ ;
        hj:
        f(hr,hi);
}

g(p,q)
{
        int gi; register int gr;
        gi = q−p;
        gr = q−p+1;
        gcnt++ ;
        gj:
        h(gr,gi);
}

f(a,b)
{
        int fi; register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++ ;
        fj:
        g(fr,fi);
}

main()
{
        f(1,1);
}
```

Figure 4: ADB output for C program of Figure 3

```
adb
$c
~h(04452,04451)
~g(04453,011124)
~f(02,04451)
~h(04450,04447)
~g(04451,011120)
~f(02,04447)
~h(04446,04445)
~g(04447,011114)
~f(02,04445)
~h(04444,04443)
HIT DEL KEY
adb
,5$C
~h(04452,04451)
            x:          04452
            y:          04451
            hi:         ?
~g(04453,011124)
            p:          04453
            q:          011124
            gi:         04451
            gr:         ?
~f(02,04451)
            a:          02
            b:          04451
            fi:         011124
            fr:         04453
~h(04450,04447)
            x:          04450
            y:          04447
            hi:         04451
            hr:         02
~g(04451,011120)
            p:          04451
            q:          011120
            gi:         04447
            gr:         04450
fcnt/d
_fcnt:          1173
gcnt/d
_gcnt:          1173
hcnt/d
_hcnt:          1172
h.x/d
022004:         2346
$q
```

Figure 5: C program to decode tabs

```
#define MAXLINE 80
#define YES      1
#define NO       0
#define TABSP    8

char    input[] "data";
char    ibuf[518];
int     tabs[MAXLINE];

main()
{
        int col, *ptab;
        char c;

        ptab = tabs;
        settab(ptab);       /*Set initial tab stops */
        col = 1;
        if(fopen(input,ibuf) < 0) {
                printf("%s : not found\n",input);
                exit(8);
        }
        while((c = getc(ibuf)) != −1) {
                switch(c) {
                        case '\t':  /* TAB */
                                while(tabpos(col) != YES) {
                                        putchar(' ');       /* put BLANK */
                                        col++ ;
                                }
                                break;
                        case '\n': /*NEWLINE */
                                putchar('\n');
                                col = 1;
                                break;
                        default:
                                putchar(c);
                                col++ ;
                }
        }
}

/* Tabpos return YES if col is a tab stop */
tabpos(col)
int col;
{
        if(col > MAXLINE)
                return(YES);
        else
                return(tabs[col]);
}

/* Settab - Set initial tab stops */
settab(tabp)
int *tabp;
{
        int i;

        for(i = 0; i<= MAXLINE; i++)
                (i%TABSP) ? (tabs[i] = NO) : (tabs[i] = YES);
}
```

Figure 6a: ADB output for C program of Figure 5

```
adb a.out –
settab+4:b
fopen+4:b
getc+4:b
tabpos+4:b
$b
breakpoints
count     bkpt              command
1         ~tabpos+04
1         _getc+04
1         _fopen+04
1         ~settab+04
settab,5?ia
~settab:          jsr       r5,csv
~settab+04:       tst       –(sp)
~settab+06:       clr       0177770(r5)
~settab+012:      cmp       $0120,0177770(r5)
~settab+020:      blt       ~settab+076
~settab+022:
settab,5?i
~settab:          jsr       r5,csv
                  tst       –(sp)
                  clr       0177770(r5)
                  cmp       $0120,0177770(r5)
                  blt       ~settab+076
:r
a.out: running
breakpoint        ~settab+04:       tst       –(sp)
settab+4:d
:c
a.out: running
breakpoint        _fopen+04:        mov       04(r5),nulstr+012
$C
_fopen(02302,02472)
~main(01,0177770)
          col:        01
          c:          0
          ptab:       03500
tabs,3/8o
03500:            01      0       0       0       0       0       0       0
                  01      0       0       0       0       0       0       0
                  01      0       0       0       0       0       0       0
```

Figure 6b: ADB output for C program of Figure 5

```
:c
a.out: running
breakpoint      _getc+04:       mov     04(r5),r1
ibuf+6/20c
__cleanu+0202:                  This    is      a test    of
:c
a.out: running
breakpoint      ~tabpos+04:     cmp     $0120,04(r5)
tabpos+4:d
settab+4:b  settab,5?ia
settab+4:b  settab,5?ia;  0
getc+4,3:b  main.c?C;  0
settab+4:b  settab,5?ia;  ptab/o;  0
$b
breakpoints
count   bkpt            command
1       ~tabpos+04
3       _getc+04 main.c?C;0
1       _fopen+04
1       ~settab+04      settab,5?ia;ptab?o;0
~settab:        jsr     r5,csv
~settab+04:     bpt
~settab+06:     clr     0177770(r5)
~settab+012:    cmp     $0120,0177770(r5)
~settab+020:    blt     ~settab+076
~settab+022:
0177766: 0177770
0177744: @`
T0177744:       T
h0177744:       h
i0177744:       i
s0177744:       s
```

Figure 7: ADB output for C program with breakpoints

```
adb ex3 –
h+4:b hcnt/d; h.hi/; h.hr/
g+4:b gcnt/d; g.gi/; g.gr/
f+4:b fcnt/d; f.fi/; f.fr/
:r
ex3: running
_fcnt:      0
0177732:    214
symbol not found
f+4:b fcnt/d; f.a/; f.b/; f.fi/
g+4:b gcnt/d; g.p/; g.q/; g.gi/
h+4:b hcnt/d; h.x/; h.y/; h.hi/
:c
ex3: running
_fcnt:      0
0177746:    1
0177750:    1
0177732:    214
_gcnt:      0
0177726:    2
0177730:    3
0177712:    214
_hcnt:      0
0177706:    2
0177710:    1
0177672:    214
_fcnt:      1
0177666:    2
0177670:    3
0177652:    214
_gcnt:      1
0177646:    5
0177650:    8
0177632:    214
HIT DEL
f+4:b fcnt/d; f.a/"a = "d; f.b/"b = "d; f.fi/"fi = "d
g+4:b gcnt/d; g.p/"p = "d; g.q/"q = "d; g.gi/"gi = "d
h+4:b hcnt/d; h.x/"x = "d; h.y/"h = "d; h.hi/"hi = "d
:r
ex3: running
_fcnt:      0
0177746:    a = 1
0177750:    b = 1
0177732:    fi = 214
_gcnt:      0
0177726:    p = 2
0177730:    q = 3
0177712:    gi = 214
_hcnt:      0
0177706:    x = 2
0177710:    y = 1
0177672:    hi = 214
_fcnt:      1
0177666:    a = 2
0177670:    b = 3
0177652:    fi = 214
HIT DEL
$q
```

Figure 8: ADB address maps

407 files

```
a.out       hdr              text+data
          |_____|______________________________|
                 0                            D

core        hdr              text+data                      stack
          |_______|______________________________......|  _____________|
                   0                             D     S              E
```

410 files (shared text)

```
a.out       hdr              text                           data
          |_____|______________________________|_________________|
                 0                            T B                D

core        hdr            data                  stack
          |_______|_________________......|  ____________|
                   B                D     S             E
```

411 files (separated I and D space)

```
a.out       hdr              text                           data
          |_____|______________________________|_________________|
                 0                            T 0                D

core        hdr            data                  stack
          |_______|_________________......|  ____________|
                   0                D     S             E
```

The following *adb* variables are set.

		407	410	411
b	base of data	0	B	0
d	length of data	D	D–B	D
s	length of stack	S	S	S
t	length of text	0	T	T

Figure 9: ADB output for maps

```
adb map407 core407
$m
text map    `map407´
b1 = 0            e1      = 0256          f1 = 020
b2 = 0            e2      = 0256          f2 = 020
data map    `core407´
b1 = 0            e1      = 0300          f1 = 02000
b2 = 0175400      e2      = 0200000   f2 = 02300
$v
variables
d = 0300
m = 0407
s = 02400
$q


adb map410 core410
$m
text map    `map410´
b1 = 0            e1      = 0200          f1 = 020
b2 = 020000       e2      = 020116    f2 = 0220
data map    `core410´
b1 = 020000       e1      = 020200    f1 = 02000
b2 = 0175400      e2      = 0200000   f2 = 02200
$v
variables
b = 020000
d = 0200
m = 0410
s = 02400
t = 0200
$q


adb map411 core411
$m
text map    `map411´
b1 = 0            e1      = 0200          f1 = 020
b2 = 0            e2      = 0116          f2 = 0220
data map    `core411´
b1 = 0            e1      = 0200          f1 = 02000
b2 = 0175400      e2      = 0200000   f2 = 02200
$v
variables
d = 0200
m = 0411
s = 02400
t = 0200
$q
```

Figure 10: Simple C program for illustrating formatting and patching

```
char    str1[]  "This is a character string";
int     one     1;
int     number  456;
long    lnum    1234;
float   fpt     1.25;
char    str2[]  "This is the second character string";
main()
{
        one = 2;
}
```

Figure 11: ADB output illustrating fancy formats

```
adb map410 core410
<b,–1/8ona
020000:         0       064124  071551  064440  020163  020141  064143  071141

_str1+016:  061541  062564  020162  072163  064562  063556  0       02

_number:
_number:    0710 0  02322   040240  0       064124  071551  064440

_str2+06:   020163  064164  020145  062563  067543  062156  061440  060550

_str2+026:  060562  072143  071145  071440  071164  067151  0147 0

savr5+02:   0   0   0   0   0   0   0   0

<b,20/4o4^8Cn
020000:         0   064124  071551  064440  @`@`This i
            020163  020141  064143  071141  s a char
            061541  062564  020162  072163  acter st
            064562  063556  0       02      ring@`@`@b@`

_number:    0710 0  02322   040240  H@a@`@`R@d @@
            0   064124  071551  064440  @`@`This i
            020163  064164  020145  062563  s the se
            067543  062156  061440  060550  cond cha
            060562  072143  071145  071440  racter s
            071164  067151  0147 0  tring@`@`@`
            0   0   0   0   @`@`@`@`@`@`@`@`
            0   0   0   0   @`@`@`@`@`@`@`@`
data address not found
<b,20/4o4^8t8cna
020000:         0   064124  071551  064440      This i
_str1+06:   020163  020141  064143  071141      s a char
_str1+016:  061541  062564  020162  072163      acter st
_str1+026:  064562  063556  0   02      ring
_number:
_number:    0710 0  02322   040240      HR
_fpt+02:    0   064124  071551  064440      This i
_str2+06:   020163  064164  020145  062563      s the se
_str2+016:  067543  062156  061440  060550      cond cha
_str2+026:  060562  072143  071145  071440      racter s
_str2+036:  071164  067151  0147 0      tring
savr5+02:   0   0   0   0
savr5+012:  0   0   0   0
data address not found
<b,10/2b8t^2cn
020000:         0   0

_str1:      0124 0150   Th
            0151 0163   is
            040  0151    i
            0163 040    s
            0141 040    a
            0143 0150   ch
            0141 0162   ar
            0141 0143   ac
            0164 0145   te
$Q
```

Figure 12: Directory and inode dumps

```
adb dir –
=nt"Inode"t"Name"
0,–1?ut14cn

        Inode Name
0:      652   .
        82    ..
        5971  cap.c
        5323  cap
        0     pp
```

```
adb /dev/src –
02000>b
?m<b
new map     `/dev/src´
b1 = 02000    e1    = 0100000000    f1 = 0
b2 = 0        e2    = 0             f2 = 0
$v
variables
b = 02000
<b,–1?"flags"8ton"links,uid,gid"8t3bn"size"8tbrdn"addr"8t8un"times"8t2Y2na
02000:      flags  073145
        links,uid,gid    0163  0164  0141
        size   0162  10356
        addr   28770    8236  25956    27766    25455    8236  25956    25206
        times 1976 Feb 5 08:34:56    1975 Dec 28 10:55:15

02040:      flags  024555
        links,uid,gid    012   0163  0164
        size   0162  25461
        addr   8308  30050    8294  25130    15216    26890    29806    10784
        times 1976 Aug 17 12:16:51   1976 Aug 17 12:16:51

02100:      flags  05173
        links,uid,gid    011   0162  0145
        size   0147  29545
        addr   25972    8306  28265    8308  25642    15216    2314  25970
        times 1977 Apr 2 08:58:01    1977 Feb 5 10:21:44
```

ADB Summary

Command Summary

a) formatted printing

? *format*	print from *a.out* file according to *format*
/ *format*	print from *core* file according to *format*
= *format*	print the value of *dot*
?w expr	write expression into *a.out* file
/w expr	write expression into *core* file
?l expr	locate expression in *a.out* file

b) breakpoint and program control

:b	set breakpoint at *dot*
:c	continue running program
:d	delete breakpoint
:k	kill the program being debugged
:r	run *a.out* file under ADB control
:s	single step

c) miscellaneous printing

$b	print current breakpoints
$c	C stack trace
$e	external variables
$f	floating registers
$m	print ADB segment maps
$q	exit from ADB
$r	general registers
$s	set offset for symbol match
$v	print ADB variables
$w	set output line width

d) calling the shell

!	call *shell* to read rest of line

e) assignment to variables

>*name*	assign dot to variable or register *name*

Format Summary

a	the value of dot
b	one byte in octal
c	one byte as a character
d	one word in decimal
f	two words in floating point
i	PDP 11 instruction
o	one word in octal
n	print a newline
r	print a blank space
s	a null terminated character string
*n***t**	move to next *n* space tab
u	one word as unsigned integer
x	hexadecimal
Y	date
ˆ	backup dot
"..."	print string

Expression Summary

a) expression components

decimal integer	e.g. 256
octal integer	e.g. 0277
hexadecimal	e.g. #ff
symbols	e.g. flag _main main.argc
variables	e.g. <b
registers	e.g. <pc <r0
(expression)	expression grouping

b) dyadic operators

+	add
–	subtract
*****	multiply
%	integer division
&	bitwise and
|	bitwise or
#	round up to the next multiple

c) monadic operators

~	not
*****	contents of location
–	integer negate