UNIX Programmer's Supplementary Documents Volume 2 (PS2)

4.3 Berkeley Software Distribution
490147 Rev. A

July 1987

Integrated Solutions
1140 Ringwood Court
San Jose, CA 95131
(408) 943-1902

This manual reflects system enhancements made at Berkeley and sponsored in part by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871 monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in these documents are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Government.

UNIX Programmer's Supplementary Documents, Volume 2 (PS2)

4.3 Berkeley Software Distribution, Virtual VAX–11 Version

April, 1986

These two volumes contain documents that supplement the manual pages in *The Unix Programmer's Reference Manual* for the Virtual VAX-11 version of the system as distributed by U.C. Berkeley.

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.

* Copyright 1974, Association for Computing Machinery, Inc., reprinted by permission. 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.

† UNIX is a trademark of Bell Laboratories.

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

C compiler
Text editor based on QED
qed lampson
Assembler, linking loader, symbolic debugger
Phototypesetting and equation setting programs
cherry kernighan typesetting mathematics cacm
kernighan lesk ossanna document preparation bstj
%Q This issue
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, dolotta mashey workbench software engineering dolotta haight mashey workbench bstj %Q This issue for example. There are also much smaller, though somewhat restricted, versions of the system. lycklama microprocessor bstj %Q This issue

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. c programming language kernighan ritchie prentice-hall 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." aleph null software practice

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 **arg_1, arg_2, . . . , 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, bourne shell bstj %Q This issue 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. lampson deutsch 930 manual 1965 system preliminary On a number of points we were influenced by Multics, which suggested the particular form of the I/O system calls multics input output feiertag organick 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. bobrow burchfiel tenex

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. $LIST$

UNIX/32V — Summary

March 9, 1979

A. What's new: highlights of the UNIX†/32V System

32-bit world. UNIX/32V handles 32-bit addresses and 32-bit data. Devices are addressable to 2^{31} bytes, files to 2^{30} bytes.

Portability. Code of the operating system and most utilities has been extensively revised to minimize its dependence on particular hardware. UNIX/32V is highly compatible with UNIX version 7.

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.

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 UNIX/32V operating system runs on a DEC VAX-11/780* with at least the following equipment:

memory: 256K bytes or more.

disk: RP06, RM03, or equivalent.

tape: any 9-track MASSBUS-compatible tape drive.

The following equipment is strongly recommended:

communications controller such as DZ11 or DL11.

full duplex 96-character ASCII terminals.

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/32V, and to keep all source on line. More memory will be needed to handle a large number of users, big data bases, diversified complements of devices, or large programs. The resident code occupies 40-55K bytes depending on configuration; system data also occupies 30-55K bytes.

C. Software

Most of the programs available as UNIX/32V 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, and a compiler for the programming language C—enough software to write and run new applications and to maintain or modify UNIX/32V 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:

- O Reentrant code for user processes.
- O "Group" access permissions for cooperative projects, with overlapping memberships.
- O Alarm-clock timeouts.
- O Timer-interrupt sampling and interprocess monitoring for debugging and measurement.
- O 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:

- O Asynchronous interfaces: DZ11, DL11. Support for most common ASCII terminals.
- O Automatic calling unit interface: DN11.

*VAX is a Trademark of Digital Equipment Corporation.

O Printer/plotter: Versatek.
O Magnetic tape: TE16.
O Pack type disk: RP06, RM03; minimum-latency seek scheduling.
O Physical memory of VAX-11, or mapped memory in resident system.
O Null device.
O Recipies are supplied to aid the construction of drivers for:
 Asynchronous interface: DH11.
 Synchronous interface: DU11.
 DECtape: TC11.
 Fixed head disk: RS11, RS03 and RS04.
 Cartridge-type disk: RK05.
 Phototypesetter: Graphic Systems System/1 through DR11C.

☐ BOOT Procedures to get UNIX/32V started.

1.2. User Access Control

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

☐ PASSWD Change a password.
O User can change his own password.
O 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.
O Half vs. full duplex.
O Carriage return+line feed vs. newline.
O Interpretation of tabs.
O Parity.
O Mapping of upper case to lower.
O Raw vs. edited input.
O 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/32V'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/32V.
☐ 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. ○ Creates object program consisting of code, normally read-only and sharable, initialized data or read-write code,

uninitialized data.
O Relocatable object code is directly executable without further transformation.
O Object code normally includes a symbol table.
O "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.
O Buffered character-by-character I/O.
O Formatted input and output conversion (SCANF and PRINTF) for standard input and output, files, in-memory conversion.
O Storage allocator.
O Time conversions.
O Number conversions.
O Password encryption.
O Quicksort.
O Random number generator.
O Mathematical function library, including trigonometric functions and inverses, exponential, logarithm, square root, bessel functions.

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

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

☐ LD — Link edit. Combine relocatable object files. Insert required routines from specified libraries.
O Resulting code is sharable by default.

☐ 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 memory 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.
O 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/32V Programmer's Manual

☐ Manual — Machine-readable version of the UNIX/32V 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/32V 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/32V 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 VAX-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.
- O Compatible with C and supporting tools at object level.
- O Optional source compatibility with Fortran 66.
- O Free format source.
- O Optional subscript-range checking, detection of uninitialized variables.
- O 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.
- O Compound statements.
- O If-else, do, for, while, repeat-until, break, next statements.
- O Symbolic constants.
- O File insertion.
- O Free format source
- O Translation of relationals like >, >=.
- O Produces genuine Fortran to carry away.
- O 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

☐ DC — Interactive programmable desk calculator. Has named storage locations as well as conventional stack for holding integers or programs.
- O Unlimited precision decimal arithmetic.
- O Appropriate treatment of decimal fractions.
- O Arbitrary input and output radices, in particular binary, octal, decimal and hexadecimal.
- O 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.
- O All the capabilities of DC with a high-level syntax.
- O Arrays and recursive functions.
- O Immediate evaluation of expressions and evaluation of functions upon call.
- O Arbitrary precision elementary functions: exp, sin, cos, atan.
- O Go-to-less programming.

2.4. Macroprocessing

☐ M4 — A general purpose macroprocessor.
- O Stream-oriented, recognizes macros anywhere in text.
- O Syntax fits with functional syntax of most higher-level languages.
- O 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.

O BNF syntax specifications.
O Precedence relations.
O 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.
O Full regular expression, plus left and right context dependence.
O 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.
O 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.
O Add, delete, change, copy, move or join lines.
O Permute or split contents of a line.
O Replace one or all instances of a pattern within a line.
O Combine or split files.
O Escape to Shell (command language) during editing.
O Do any of above operations on every pattern-selected line in a given range.
O 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.
O 25,000-word list includes proper names.
O Handles common prefixes and suffixes.
O Collects words to help tailor local spelling lists.

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

☐ CRYPT — Encrypt and decrypt files for security.

3.2. Document Formatting

☐ 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 are capable of elaborate feats of formatting, when appropriately programmed. TROFF and NROFF accept the same input language.
O Completely definable page format keyed to dynamically planted "interrupts" at specified lines.
O Maintains several separately definable typesetting environments (e.g., one for body text, one for footnotes, and one for unusually elaborate headings).
O Arbitrary number of output pools can be combined at will.
O Macros with substitutable arguments, and macros invocable in mid-line.
O Computation and printing of numerical quantities.
O Conditional execution of macros.
O Tabular layout facility.
O Positions expressible in inches, centimeters, ems, points, machine units or arithmetic combinations thereof.
O 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 postprocessor 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.
☐ 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.
□ BCD	Converts ascii to card-image form.
□ CAL	Print a calendar of specified month and year.
□ 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?
□ ARITHMETIC	Speed and accuracy test for number facts.
□ QUIZ	Test your knowledge of Shakespeare, Presidents, capitals, etc.
□ WUMP	Hunt the wumpus, thrilling search in a dangerous cave.
□ HANGMAN	Word-guessing game. Uses a dictionary supplied with SPELL.
□ FISH	Children's card-guessing game.

UNIX Programming — Second Edition

Brian W. Kernighan

Dennis M. Ritchie

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.

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.)

† UNIX is a trademark of Bell Laboratories.

```
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

```
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 predefined 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

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`.

UNIX Implementation

K. Thompson

ABSTRACT

This paper describes in high-level terms the implementation of the resident UNIX† kernel. This discussion is broken into three parts. The first part describes how the UNIX system views processes, users, and programs. The second part describes the I/O system. The last part describes the UNIX file system.

1. INTRODUCTION

The UNIX kernel consists of about 10,000 lines of C code and about 1,000 lines of assembly code. The assembly code can be further broken down into 200 lines included for the sake of efficiency (they could have been written in C) and 800 lines to perform hardware functions not possible in C.

This code represents 5 to 10 percent of what has been lumped into the broad expression "the UNIX operating system." The kernel is the only UNIX code that cannot be substituted by a user to his own liking. For this reason, the kernel should make as few real decisions as possible. This does not mean to allow the user a million options to do the same thing. Rather, it means to allow only one way to do one thing, but have that way be the least-common divisor of all the options that might have been provided.

What is or is not implemented in the kernel represents both a great responsibility and a great power. It is a soap-box platform on "the way things should be done." Even so, if "the way" is too radical, no one will follow it. Every important decision was weighed carefully. Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized.

2. PROCESS CONTROL

In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process calls the system as a subroutine. At some point in this call, there is a distinct switch of environments. After this, the process is said to be a system process. In the normal definition of processes, the user and system processes are different phases of the same process (they never execute simultaneously). For protection, each system process has its own stack.

The user process may execute from a read-only text segment, which is shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit comes from the fact that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory, that is, when there is enough memory to keep waiting processes loaded.

All current read-only text segments in the system are maintained from the *text table*. A text table entry holds the location of the text segment on secondary memory. If the segment is loaded, that table also holds the primary memory location and the count of the number of processes sharing this entry. When this count is reduced to zero, the entry is freed along with any primary and secondary memory holding the

† UNIX is a trademark of Bell Laboratories.

segment. When a process first executes a shared-text segment, a text table entry is allocated and the segment is loaded onto secondary memory. If a second process executes a text segment that is already allocated, the entry reference count is simply incremented.

A user process has some strictly private read-write data contained in its data segment. As far as possible, the system does not use the user's data segment to hold system data. In particular, there are no I/O buffers in the user address space.

The user data segment has two growing boundaries. One, increased automatically by the system as a result of memory faults, is used for a stack. The second boundary is only grown (or shrunk) by explicit requests. The contents of newly allocated primary memory is initialized to zero.

Also associated and swapped with a process is a small fixed-size system data segment. This segment contains all the data about the process that the system needs only when the process is active. Examples of the kind of data contained in the system data segment are: saved central processor registers, open file descriptors, accounting information, scratch data area, and the stack for the system phase of the process. The system data segment is not addressable from the user process and is therefore protected.

Last, there is a process table with one entry per process. This entry contains all the data needed by the system when the process is *not* active. Examples are the process's name, the location of the other segments, and scheduling information. The process table entry is allocated when the process is created, and freed when the process terminates. This process entry is always directly addressable by the kernel.

Figure 1 shows the relationships between the various process control data. In a sense, the process table is the definition of all processes, because all the data associated with a process may be accessed starting from the process table entry.

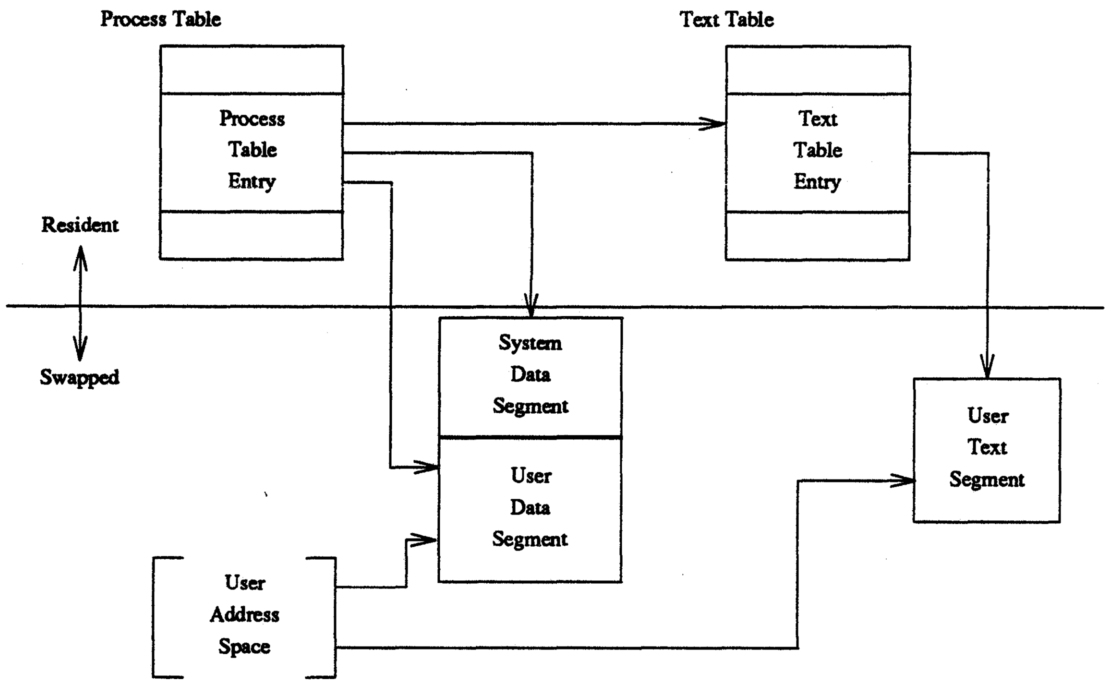

Fig. 1—Process control data structure.

2.1. Process creation and program execution

Processes are created by the system primitive **fork**. The newly created process (child) is a copy of the original process (parent). There is no detectable sharing of primary memory between the two processes. (Of course, if the parent process was executing from a read-only text segment, the child will share the text segment.) Copies of all writable data segments are made for the child process. Files that were open before the **fork** are truly shared after the **fork**. The processes are informed as to their part in the relationship to allow them to select their own (usually non-identical) destiny. The parent may **wait** for the termination of any of its children.

A process may **exec** a file. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. Doing an **exec** does *not* change processes; the process that did the **exec** persists, but after the **exec** it is executing a different program. Files that were open before the **exec** remain open after the **exec**.

If a program, say the first pass of a compiler, wishes to overlay itself with another program, say the second pass, then it simply **execs** the second program. This is analogous to a "goto." If a program wishes to regain control after **exec**ing a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a "call." Breaking up the call into a binding followed by a transfer is similar to the subroutine linkage in SL-5.[1] griswold hanson sl5 overview

2.2. Swapping

The major data associated with a process (the user data segment, the system data segment, and the text segment) are swapped to and from secondary memory, as needed. The user data segment and the system data segment are kept in contiguous primary memory to reduce swapping latency. (When low-latency devices, such as bubbles, CCDs, or scatter/gather devices, are used, this decision will have to be reconsidered.) Allocation of both primary and secondary memory is performed by the same simple first-fit algorithm. When a process grows, a new piece of primary memory is allocated. The contents of the old memory is copied to the new memory. The old memory is freed and the tables are updated. If there is not enough primary memory, secondary memory is allocated instead. The process is swapped out onto the secondary memory, ready to be swapped in with its new size.

One separate process in the kernel, the swapping process, simply swaps the other processes in and out of primary memory. It examines the process table looking for a process that is swapped out and is ready to run. It allocates primary memory for that process and reads its segments into primary memory, where that process competes for the central processor with other loaded processes. If no primary memory is available, the swapping process makes memory available by examining the process table for processes that can be swapped out. It selects a process to swap out, writes it to secondary memory, frees the primary memory, and then goes back to look for a process to swap in.

Thus there are two specific algorithms to the swapping process. Which of the possibly many processes that are swapped out is to be swapped in? This is decided by secondary storage residence time. The one with the longest time out is swapped in first. There is a slight penalty for larger processes. Which of the possibly many processes that are loaded is to be swapped out? Processes that are waiting for slow events (i.e., not currently running or waiting for disk I/O) are picked first, by age in primary memory, again with size penalties. The other processes are examined by the same age algorithm, but are not taken out unless they are at least of some age. This adds hysteresis to the swapping and prevents total thrashing.

These swapping algorithms are the most suspect in the system. With limited primary memory, these algorithms cause total swapping. This is not bad in itself, because the swapping does not impact the execution of the resident processes. However, if the swapping device must also be used for file storage, the swapping traffic severely impacts the file system traffic. It is exactly these small systems that tend to double usage of limited disk resources.

2.3. Synchronization and scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. By convention, events are chosen to be addresses of tables associated with those events. For example, a process that is waiting for any of its children to terminate will wait for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by its parent's process table entry. Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. This differs considerably from Dijkstra's P and V synchronization operations,[2] dijkstra sequential processes

1968 in that no memory is associated with events. Thus there need be no allocation of events prior to their use. Events exist simply by being used.

On the negative side, because there is no memory associated with events, no notion of "how much" can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event would be signaled. All the competing processes would then wake up to fight over the new memory. (In reality, the swapping process is the only process that waits for primary memory to become available.)

If an event occurs between the time a process decides to wait for that event and the time that process enters the wait state, then the process will wait on an event that has already happened (and may never happen again). This race condition happens because there is no memory associated with the event to indicate that the event has occurred; the only action of an event is to change a set of processes from wait state to run state. This problem is relieved largely by the fact that process switching can only occur in the kernel by explicit calls to the event-wait mechanism. If the event in question is signaled by another process, then there is no problem. But if the event is signaled by a hardware interrupt, then special care must be taken. These synchronization races pose the biggest problem when UNIX is adapted to multiple-processor configurations.[2] hawley meyer multiprocessing unix

The event-wait code in the kernel is like a co-routine linkage. At any time, all but one of the processes has called event-wait. The remaining process is the one currently executing. When it calls event-wait, a process whose event has been signaled is selected and that process returns from its call to event-wait.

Which of the runable processes is to run next? Associated with each process is a priority. The priority of a system process is assigned by the code issuing the wait on an event. This is roughly equivalent to the response that one would expect on such an event. Disk events have high priority, teletype events are low, and time-of-day events are very low. (From observation, the difference in system process priorities has little or no performance impact.) All user-process priorities are lower than the lowest system priority. User-process priorities are assigned by an algorithm based on the recent ratio of the amount of compute time to real time consumed by the process. A process that has used a lot of compute time in the last real-time unit is assigned a low user priority. Because interactive processes are characterized by low ratios of compute to real time, interactive response is maintained without any special arrangements.

The scheduling algorithm simply picks the process with the highest priority, thus picking all system processes first and user processes second. The compute-to-real-time ratio is updated every second. Thus, all other things being equal, looping user processes will be scheduled round-robin with a 1-second quantum. A high-priority process waking up will preempt a running, low-priority process. The scheduling algorithm has a very desirable negative feedback character. If a process uses its high priority to hog the computer, its priority will drop. At the same time, if a low-priority process is ignored for a long time, its priority will rise.

3. I/O SYSTEM

The I/O system is broken into two completely separate systems: the block I/O system and the character I/O system. In retrospect, the names should have been "structured I/O" and "unstructured I/O," respectively; while the term "block I/O" has some meaning, "character I/O" is a complete misnomer.

Devices are characterized by a major device number, a minor device number, and a class (block or character). For each class, there is an array of entry points into the device drivers. The major device number is used to index the array when calling the code for a particular device driver. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

The use of the array of entry points (configuration table) as the only connection between the system code and the device drivers is very important. Early versions of the system had a much less formal connection with the drivers, so that it was extremely hard to handcraft differently configured systems. Now it is possible to create new device drivers in an average of a few hours. The configuration table in most cases is created automatically by a program that reads the system's parts list.

3.1. Block I/O system

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 bytes each. The blocks are uniformly addressed 0, 1, ... up to the size of the device. The block device driver has the job of emulating this model on a physical device.

The block I/O devices are accessed through a layer of buffering software. The system maintains a list of buffers (typically between 10 and 70) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data are made available. Write requests are handled in an analogous manner. The correct buffer is found and relabeled if necessary. The write is performed simply by marking the buffer as "dirty." The physical I/O is then deferred until the buffer is renamed.

The benefits in reduction of physical I/O of this scheme are substantial, especially considering the file system implementation. There are, however, some drawbacks. The asynchronous nature of the algorithm makes error reporting and meaningful user error handling almost impossible. The cavalier approach to I/O error handling in the UNIX system is partly due to the asynchronous nature of the block I/O system. A second problem is in the delayed writes. If the system stops unexpectedly, it is almost certain that there is a lot of logically complete, but physically incomplete, I/O in the buffers. There is a system primitive to flush all outstanding I/O activity from the buffers. Periodic use of this primitive helps, but does not solve, the problem. Finally, the associativity in the buffers can alter the physical I/O sequence from that of the logical I/O sequence. This means that there are times when data structures on disk are inconsistent, even though the software is careful to perform I/O in the correct order. On non-random devices, notably magnetic tape, the inversions of writes can be disastrous. The problem with magnetic tapes is "cured" by allowing only one outstanding write request per drive.

3.2. Character I/O system

The character I/O system consists of all devices that do not fall into the block I/O model. This includes the "classical" character devices such as communications lines, paper tape, and line printers. It also includes magnetic tape and disks when they are not used in a stereotyped way, for example, 80-byte physical records on tape and track-at-a-time disk copies. In short, the character I/O interface means "everything other than block." I/O requests from the user are sent to the device driver essentially unaltered. The implementation of these requests is, of course, up to the device driver. There are guidelines and conventions to help the implementation of certain types of device drivers.

3.2.1. Disk drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing such a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also insures that the user is not swapped during this I/O transaction. Thus by implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

3.2.2. Character lists

Real character-oriented devices may be implemented using the common code to handle character lists. A character list is a queue of characters. One routine puts a character on a queue. Another gets a character from a queue. It is also possible to ask how many characters are currently on a queue. Storage for all queues in the system comes from a single common pool. Putting a character on a queue will allocate space from the common pool and link the character onto the data structure defining the queue. Getting a character from a queue returns the corresponding space to the pool.

A typical character-output device (paper tape punch, for example) is implemented by passing characters from the user onto a character queue until some maximum number of characters is on the queue. The I/O is prodded to start as soon as there is anything on the queue and, once started, it is sustained by hardware completion interrupts. Each time there is a completion interrupt, the driver gets the next character from the queue and sends it to the hardware. The number of characters on the queue is checked and, as the count falls through some intermediate level, an event (the queue address) is signaled. The process that is passing characters from the user to the queue can be waiting on the event, and refill the queue to its maximum when the event occurs.

A typical character input device (for example, a paper tape reader) is handled in a very similar manner.

Another class of character devices is the terminals. A terminal is represented by three character queues. There are two input queues (raw and canonical) and an output queue. Characters going to the output of a terminal are handled by common code exactly as described above. The main difference is that there is also code to interpret the output stream as ASCII characters and to perform some translations, e.g., escapes for deficient terminals. Another common aspect of terminals is code to insert real-time delay after certain control characters.

Input on terminals is a little different. Characters are collected from the terminal and placed on a raw input queue. Some device-dependent code conversion and escape interpretation is handled here. When a line is complete in the raw queue, an event is signaled. The code catching this signal then copies a line from the raw queue to a canonical queue performing the character erase and line kill editing. User read requests on terminals can be directed at either the raw or canonical queues.

3.2.3. Other character devices

Finally, there are devices that fit no general category. These devices are set up as character I/O drivers. An example is a driver that reads and writes unmapped primary memory as an I/O device. Some devices are too fast to be treated a character at time, but do not fit the disk I/O mold. Examples are fast communications lines and fast line printers. These devices either have their own buffers or "borrow" block I/O buffers for a while and then give them back.

4. THE FILE SYSTEM

In the UNIX system, a file is a (one-dimensional) array of bytes. No other structure of files is implied by the system. Files are attached anywhere (and possibly multiply) onto a hierarchy of directories. Directories are simply files that users cannot write. For a further discussion of the external view of files and directories. See Ref. 3

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a "disk" is a randomly addressable array of 512-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called "super-block." This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the i-list, a list of file definitions. Each file definition is a 64-byte structure, called an i-node. The offset of a particular i-node within the i-list is called its i-number. The combination of device name (major and minor numbers) and i-number serves to uniquely name a particular file. After the i-list, and to the end of the disk, come free storage blocks that are available for the contents of files.

The free space on a disk is maintained by a linked list of available disk blocks. Every block in this chain contains a disk address of the next block in the chain. The remaining space contains the address of up to 50 disk blocks that are also free. Thus with one I/O operation, the system obtains 50 free blocks and a pointer where to find more. The disk allocation algorithms are very straightforward. Since all allocation is in fixed-size blocks and there is strict accounting of space, there is no need to compact or garbage collect. However, as disk space becomes dispersed, latency gradually increases. Some installations choose to occasionally compact disk space to reduce latency.

An i-node contains 13 disk addresses. The first 10 of these addresses point directly at the first 10 blocks of a file. If a file is larger than 10 blocks (5,120 bytes), then the eleventh address points at a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this (70,656 bytes), then the twelfth block points at up to 128 blocks, each pointing to 128 blocks of the file. Files yet larger (8,459,264 bytes) use the thirteenth address for a "triple indirect" address. The algorithm ends here with the maximum file size of 1,082,201,087 bytes.

A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file, the directory. A directory is accessed exactly as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an i-number. The root of the hierarchy is at a known i-number (*viz.*, 2). The file system structure allows an arbitrary, directed graph of directories with regular files linked in at arbitrary places in this graph. In fact, very early UNIX systems used such a structure. Administration of such a structure became so chaotic that later systems were restricted to a directory tree. Even now, with regular files linked multiply into arbitrary places in the tree, accounting for space has become a problem. It may become necessary to restrict the entire structure to a tree, and allow a new form of linking that is subservient to the tree structure.

The file system allows easy creation, easy removal, easy random accessing, and very easy space allocation. With most physical addresses confined to a small contiguous section of disk, it is also easy to dump, restore, and check the consistency of the file system. Large files suffer from indirect addressing, but the cache prevents most of the implied physical I/O without adding much execution. The space overhead properties of this scheme are quite good. For example, on one particular file system, there are 25,000 files containing 130M bytes of data-file content. The overhead (i-node, indirect blocks, and last block breakage) is about 11.5M bytes. The directory structure to support these files has about 1,500 directories containing 0.6M bytes of directory content and about 0.5M bytes of overhead in accessing the directories. Added up any way, this comes out to less than a 10 percent overhead for actual stored data. Most systems have this much overhead in padded trailing blanks alone.

4.1. File system implementation

Because the i-node defines a file, the implementation of the file system centers around access to the i-node. The system maintains a table of all active i-nodes. As a new file is accessed, the system locates the corresponding i-node, allocates an i-node table entry, and reads the i-node into primary memory. As in the buffer cache, the table entry is considered to be the current version of the i-node. Modifications to the i-node are made to the table entry. When the last access to the i-node goes away, the table entry is copied back to the secondary store i-list and the table entry is freed.

All I/O operations on files are carried out with the aid of the corresponding i-node table entry. The accessing of a file is a straightforward implementation of the algorithms mentioned previously. The user is not aware of i-nodes and i-numbers. References to the file system are made in terms of path names of the directory tree. Converting a path name into an i-node table entry is also straightforward. Starting at some known i-node (the root or the current directory of some process), the next component of the path name is searched by reading the directory. This gives an i-number and an implied device (that of the directory). Thus the next i-node table entry can be accessed. If that was the last component of the path name, then this i-node is the result. If not, this i-node is the directory needed to look up the next component of the path name, and the algorithm is repeated.

The user process accesses the file system with certain primitives. The most common of these are **open, create, read, write, seek,** and **close**. The data structures maintained are shown in Fig. 2. In the system data segment associated with a user, there is room for some (usually between 10 and 50) open files. This open file table consists of pointers that can be used to access corresponding i-node table entries. Associated with each of these open files is a current I/O pointer. This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the I/O pointer. The user usually thinks of the file as sequential with the I/O pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the I/O pointer before reads/writes.

With file sharing, it is necessary to allow related processes to share a common I/O pointer and yet have separate I/O pointers for independent processes that access the same file. With these two conditions, the I/O pointer cannot reside in the i-node table nor can it reside in the list of open files for the process. A new table (the open file table) was invented for the sole purpose of holding the I/O pointer. Processes that

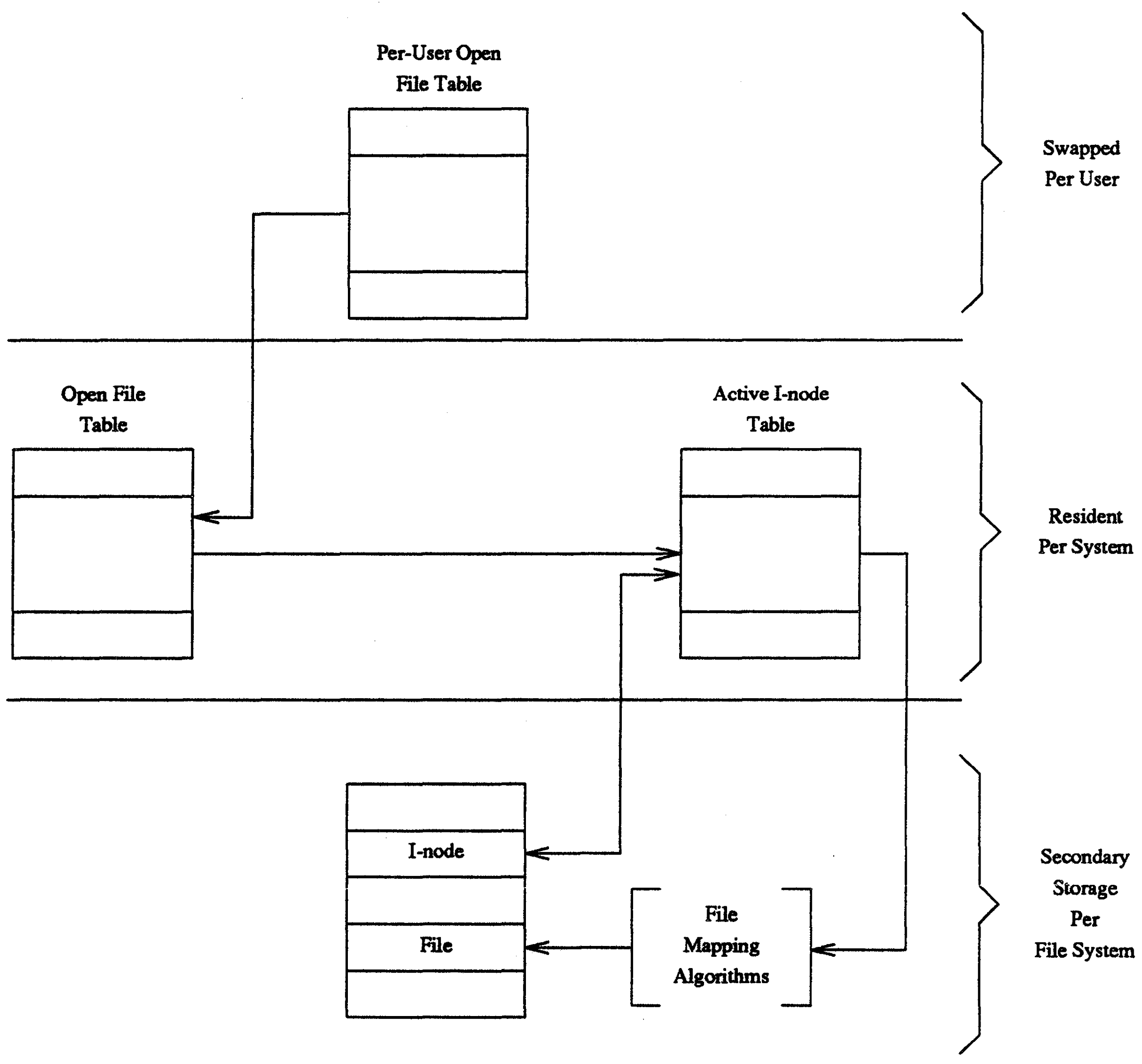

Fig. 2—File system data structure.

share the same open file (the result of **forks**) share a common open file table entry. A separate open of the same file will only share the i-node table entry, but will have distinct open file table entries.

The main file system primitives are implemented as follows. **open** converts a file system path name into an i-node table entry. A pointer to the i-node table entry is placed in a newly created open file table entry. A pointer to the file table entry is placed in the system data segment for the process. **create** first creates a new i-node entry, writes the i-number into a directory, and then builds the same structure as for an **open**. **read** and **write** just access the i-node entry as described above. **seek** simply manipulates the I/O pointer. No physical seeking is done. **close** just frees the structures built by **open** and **create**. Reference counts are kept on the open file table entries and the i-node table entries to free these structures after the last reference goes away. **unlink** simply decrements the count of the number of directories pointing at the given i-node. When the last reference to an i-node table entry goes away, if the i-node has no directories pointing to it, then the file is removed and the i-node is freed. This delayed removal of files prevents problems arising from removing active files. A file may be removed while still open. The resulting unnamed file vanishes when the file is closed. This is a method of obtaining temporary files.

There is a type of unnamed FIFO file called a **pipe.** Implementation of **pipes** consists of implied seeks before each **read** or **write** in order to implement first-in-first-out. There are also checks and synchronization to prevent the writer from grossly outproducing the reader and to prevent the reader from overtaking the writer.

4.2. Mounted file systems

The file system of a UNIX system starts with some designated block device formatted as described above to contain a hierarchy. The root of this structure is the root of the UNIX file system. A second formatted block device may be mounted at any leaf of the current hierarchy. This logically extends the current hierarchy. The implementation of mounting is trivial. A mount table is maintained containing pairs of designated leaf i-nodes and block devices. When converting a path name into an i-node, a check is made to see if the new i-node is a designated leaf. If it is, the i-node of the root of the block device replaces it.

Allocation of space for a file is taken from the free pool on the device on which the file lives. Thus a file system consisting of many mounted devices does not have a common pool of free secondary storage space. This separation of space on different devices is necessary to allow easy unmounting of a device.

4.3. Other system functions

There are some other things that the system does for the user–a little accounting, a little tracing/debugging, and a little access protection. Most of these things are not very well developed because our use of the system in computing science research does not need them. There are some features that are missed in some applications, for example, better inter-process communication.

The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features are found in most other operating systems that are missing from the UNIX kernel. For example, the UNIX kernel does not support file access methods, file disposition, file formats, file maximum size, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, login, or log-out. Many of these things are symptoms rather than features. Many of these things are implemented in user software using the kernel as a tool. A good example of this is the command language.[3] bourne shell 1978 bstj %Q This issue Each user may have his own command language. Maintenance of such code is as easy as maintaining user code. The idea of implementing ''system'' code with general user primitives comes directly from MULTICS[2].

References

1. R.E. Griswold and D.R. Hanson, "An Overview of SL5," *SIGPLAN Notices*, vol. 12, no. 4, pp. 40-50, April 1977.

2. E.W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages*, ed. F. Genuys, pp. 43-112, Academic Press, New York, 1968.

3. D.M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell Sys. Tech. J.*, vol. 57, no.6, pp. 1905-1929, 1978.

The UNIX I/O System

Dennis M. Ritchie

This paper gives an overview of the workings of the UNIX† I/O system. It was written with an eye toward providing guidance to writers of device driver routines, and is oriented more toward describing the environment and nature of device drivers than the implementation of that part of the file system which deals with ordinary files.

It is assumed that the reader has a good knowledge of the overall structure of the file system as discussed in the paper "The UNIX Time-sharing System." A more detailed discussion appears in "UNIX Implementation;" the current document restates parts of that one, but is still more detailed. It is most useful in conjunction with a copy of the system code, since it is basically an exegesis of that code.

Device Classes

There are two classes of device: *block* and *character*. The block interface is suitable for devices like disks, tapes, and DECtape which work, or can work, with addressible 512-byte blocks. Ordinary magnetic tape just barely fits in this category, since by use of forward and backward spacing any block can be read, even though blocks can be written only at the end of the tape. Block devices can at least potentially contain a mounted file system. The interface to block devices is very highly structured; the drivers for these devices share a great many routines as well as a pool of buffers.

Character-type devices have a much more straightforward interface, although more work must be done by the driver itself.

Devices of both types are named by a *major* and a *minor* device number. These numbers are generally stored as an integer with the minor device number in the low-order 8 bits and the major device number in the next-higher 8 bits; macros *major* and *minor* are available to access these numbers. The major device number selects which driver will deal with the device; the minor device number is not used by the rest of the system but is passed to the driver at appropriate times. Typically the minor number selects a subdevice attached to a given controller, or one of several similar hardware interfaces.

The major device numbers for block and character devices are used as indices in separate tables; they both start at 0 and therefore overlap.

Overview of I/O

The purpose of the *open* and *creat* system calls is to set up entries in three separate system tables. The first of these is the *u_ofile* table, which is stored in the system's per-process data area *u*. This table is indexed by the file descriptor returned by the *open* or *creat*, and is accessed during a *read*, *write*, or other operation on the open file. An entry contains only a pointer to the corresponding entry of the *file* table, which is a per-system data base. There is one entry in the *file* table for each instance of *open* or *creat*. This table is per-system because the same instance of an open file must be shared among the several processes which can result from *forks* after the file is opened. A *file* table entry contains flags which indicate whether the file was open for reading or writing or is a pipe, and a count which is used to decide when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which is used to indicate where in the file the next read or write will take place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's i-node.

†UNIX is a Trademark of Bell Laboratories.

Certain open files can be designated "multiplexed" files, and several other flags apply to such channels. In such a case, instead of an offset, there is a pointer to an associated multiplex channel table. Multiplex channels will not be discussed here.

An entry in the *file* table corresponds precisely to an instance of *open* or *creat;* if the same file is opened several times, it will have several entries in this table. However, there is at most one entry in the *inode* table for a given file. Also, a file may enter the *inode* table not only because it is open, but also because it is the current directory of some process or because it is a special file containing a currently-mounted file system.

An entry in the *inode* table differs somewhat from the corresponding i-node as stored on the disk; the modified and accessed times are not stored, and the entry is augmented by a flag word containing information about the entry, a count used to determine when it may be allowed to disappear, and the device and i-number whence the entry came. Also, the several block numbers that give addressing information for the file are expanded from the 3-byte, compressed format used on the disk to full *long* quantities.

During the processing of an *open* or *creat* call for a special file, the system always calls the device's *open* routine to allow for any special processing required (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.). However, the *close* routine is called only when the last process closes a file, that is, when the i-node table entry is being deallocated. Thus it is not feasible for a device to maintain, or depend on, a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

When a *read* or *write* takes place, the user's arguments and the *file* table entry are used to set up the variables *u.u_base, u.u_count,* and *u.u_offset* which respectively contain the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate read or write routine is called; it is responsible for transferring data and updating the count and current location appropriately as discussed below. Otherwise, the current location is used to calculate a logical block number in the file. If the file is an ordinary file the logical block number must be mapped (possibly using indirect blocks) to a physical block number; a block-type special file need not be mapped. This mapping is performed by the *bmap* routine. In any event, the resulting physical block number is used, as discussed below, to read or write the appropriate device.

Character Device Drivers

The *cdevsw* table specifies the interface routines present for character devices. Each device provides five routines: open, close, read, write, and special-function (to implement the *ioctl* system call). Any of these may be missing. If a call on the routine should be ignored, (e.g. *open* on non-exclusive devices that require no setup) the *cdevsw* entry can be given as *nulldev;* if it should be considered an error, (e.g. *write* on read-only devices) *nodev* is used. For terminals, the *cdevsw* structure also contains a pointer to the *tty* structure associated with the terminal.

The *open* routine is called each time the file is opened with the full device number as argument. The second argument is a flag which is non-zero only if the device is to be written upon.

The *close* routine is called only when the file is closed for the last time, that is when the very last process in which the file is open closes it. This means it is not possible for the driver to maintain its own count of its users. The first argument is the device number; the second is a flag which is non-zero if the file was open for writing in the process which performs the final *close*.

When *write* is called, it is supplied the device as argument. The per-user variable *u.u_count* has been set to the number of characters indicated by the user; for character devices, this number may be 0 initially. *u.u_base* is the address supplied by the user from which to start taking characters. The system may call the routine internally, so the flag *u.u_segflg* is supplied that indicates, if *on,* that *u.u_base* refers to the system address space instead of the user's.

The *write* routine should copy up to *u.u_count* characters from the user's buffer to the device, decrementing *u.u_count* for each character passed. For most drivers, which work one character at a time, the routine *cpass( )* is used to pick up characters from the user's buffer. Successive calls on it return the characters to be written until *u.u_count* goes to 0 or an error occurs, when it returns −1. *Cpass* takes care of interrogating *u.u_segflg* and updating *u.u_count.*

Write routines which want to transfer a probably large number of characters into an internal buffer may also use the routine *iomove(buffer, offset, count, flag)* which is faster when many characters must be moved. *Iomove* transfers up to *count* characters into the *buffer* starting *offset* bytes from the start of the buffer; *flag* should be *B_WRITE* (which is 0) in the write case. Caution: the caller is responsible for making sure the count is not too large and is non-zero. As an efficiency note, *iomove* is much slower if any of *buffer+offset, count* or *u.u_base* is odd.

The device's *read* routine is called under conditions similar to *write,* except that *u.u_count* is guaranteed to be non-zero. To return characters to the user, the routine *passc(c)* is available; it takes care of housekeeping like *cpass* and returns −1 as the last character specified by *u.u_count* is returned to the user; before that time, 0 is returned. *Iomove* is also usable as with *write;* the flag should be *B_READ* but the same cautions apply.

The "special-functions" routine is invoked by the *stty* and *gtty* system calls as follows: *(*p) (dev, v)* where *p* is a pointer to the device's routine, *dev* is the device number, and *v* is a vector. In the *gtty* case, the device is supposed to place up to 3 words of status information into the vector; this will be returned to the caller. In the *stty* case, *v* is 0; the device should take up to 3 words of control information from the array *u.u_arg[0...2]*.

Finally, each device should have appropriate interrupt-time routines. When an interrupt occurs, it is turned into a C-compatible call on the devices's interrupt routine. The interrupt-catching mechanism makes the low-order four bits of the "new PS" word in the trap vector for the interrupt available to the interrupt handler. This is conventionally used by drivers which deal with multiple similar devices to encode the minor device number. After the interrupt has been processed, a return from the interrupt handler will return from the interrupt itself.

A number of subroutines are available which are useful to character device drivers. Most of these handlers, for example, need a place to buffer characters in the internal interface between their "top half" (read/write) and "bottom half" (interrupt) routines. For relatively low data-rate devices, the best mechanism is the character queue maintained by the routines *getc* and *putc*. A queue header has the structure

```
struct {
        int     c_cc; /* character count */
        char    *c_cf;        /* first character */
        char    *c_cl;/* last character */
} queue;
```

A character is placed on the end of a queue by *putc(c, &queue)* where *c* is the character and *queue* is the queue header. The routine returns −1 if there is no space to put the character, 0 otherwise. The first character on the queue may be retrieved by *getc(&queue)* which returns either the (non-negative) character or −1 if the queue is empty.

Notice that the space for characters in queues is shared among all devices in the system and in the standard system there are only some 600 character slots available. Thus device handlers, especially write routines, must take care to avoid gobbling up excessive numbers of characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call *sleep(event, priority)* causes the process to wait (allowing other processes to run) until the *event* occurs; at that time, the process is marked ready-to-run and the call will return when there is no process with higher *priority.*

The call *wakeup(event)* indicates that the *event* has happened, that is, causes processes sleeping on the event to be awakened. The *event* is an arbitrary quantity agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area used by the driver, which guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened; they should check that the conditions which caused them to sleep no longer hold.

Priorities can range from 0 to 127; a higher numerical value indicates a less-favored scheduling situation. A distinction is made between processes sleeping at priority less than the parameter *PZERO* and those at numerically larger priorities. The former cannot be interrupted by signals, although it is conceivable that it may be swapped out. Thus it is a bad idea to sleep with priority less than PZERO on an event which might never occur. On the other hand, calls to *sleep* with larger priority may never return if the process is terminated by some signal in the meantime. Incidentally, it is a gross error to call *sleep* in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area "*u.*" should be touched, let alone changed, by an interrupt routine.

If a device driver wishes to wait for some event for which it is inconvenient or impossible to supply a *wakeup*, (for example, a device going on-line, which does not generally cause an interrupt), the call *sleep(&lbolt, priority)* may be given. *Lbolt* is an external cell whose address is awakened once every 4 seconds by the clock interrupt routine.

The routines *spl4( ), spl5( ), spl6( ), spl7( )* are available to set the processor priority level as indicated to avoid inconvenient interrupts from the device.

If a device needs to know about real-time intervals, then *timeout(func, arg, interval)* will be useful. This routine arranges that after *interval* sixtieths of a second, the *func* will be called with *arg* as argument, in the style *(*func)(arg)*. Timeouts are used, for example, to provide real-time delays after function characters like new-line and tab in typewriter output, and to terminate an attempt to read the 201 Dataphone *dp* if there is no response within a specified number of seconds. Notice that the number of sixtieths of a second is limited to 32767, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

The Block-device Interface

Handling of block devices is mediated by a collection of routines that manage a set of buffers containing the images of blocks of data on the various devices. The most important purpose of these routines is to assure that several processes that access the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is to increase the efficiency of the system by keeping in-core copies of blocks that are being accessed frequently. The main data base for this mechanism is the table of buffers *buf*. Each buffer header contains a pair of pointers *(b_forw, b_back)* which maintain a doubly-linked list of the buffers associated with a particular block device, and a pair of pointers *(av_forw, av_back)* which generally maintain a doubly-linked list of blocks which are "free," that is, eligible to be reallocated for another transaction. Buffers that have I/O in progress or are busy for other purposes do not appear in this list. The buffer header also contains the device and block number to which the buffer refers, and a pointer to the actual storage associated with the buffer. There is a word count which is the negative of the number of words to be transferred to or from the buffer; there is also an error byte and a residual word count used to communicate information from an I/O routine to its caller. Finally, there is a flag word with bits indicating the status of the buffer. These flags will be discussed below.

Seven routines constitute the most important part of the interface with the rest of the system. Given a device and block number, both *bread* and *getblk* return a pointer to a buffer header for the block; the difference is that *bread* is guaranteed to return a buffer actually containing the current data for the block, while *getblk* returns a buffer which contains the data in the block only if it is already in core (whether it is or not is indicated by the *B_DONE* bit; see below). In either case the buffer, and the corresponding device block, is made "busy," so that other processes referring to it are obliged to wait until it becomes free. *Getblk* is used, for example, when a block is about to be totally rewritten, so that its previous contents are not useful; still, no other process can be allowed to refer to the block until the new data is placed into it.

The *breada* routine is used to implement read-ahead. it is logically similar to *bread*, but takes as an additional argument the number of a block (on the same device) to be read asynchronously after the specifically requested block is available.

Given a pointer to a buffer, the *brelse* routine makes the buffer again available to other processes. It is called, for example, after data has been extracted following a *bread*. There are three subtly-different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list. *Bwrite* puts the buffer on the appropriate device queue, waits for the write to be done, and sets the user's error flag if required. *Bawrite* places the buffer on the device's queue, but does not wait for completion, so that errors cannot be reflected directly to the user. *Bdwrite* does not start any I/O operation at all, but merely marks the buffer so that if it happens to be grabbed from the free list to contain data from some other block, the data in it will first be written out.

Bwrite is used when one wants to be sure that I/O takes place correctly, and that errors are reflected to the proper user; it is used, for example, when updating i-nodes. *Bawrite* is useful when more overlap is desired (because no wait is required for I/O to finish) but when it is reasonably certain that the write is really required. *Bdwrite* is used when there is doubt that the write is needed at the moment. For example, *bdwrite* is called when the last byte of a *write* system call falls short of the end of a block, on the assumption that another *write* will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, *bawrite* is called, since probably the block will not be accessed again soon and one might as well start the writing process as soon as possible.

In any event, notice that the routines *getblk* and *bread* dedicate the given block exclusively to the use of the caller, and make others wait, while one of *brelse, bwrite, bawrite,* or *bdwrite* must eventually be called to free the block for use by others.

As mentioned, each buffer header contains a flag word which indicates the status of the buffer. Since they provide one important channel for information between the drivers and the block I/O system, it is important to understand these flags. The following names are manifest constants which select the associated flag bits.

B_READ This bit is set when the buffer is handed to the device strategy routine (see below) to indicate a read operation. The symbol *B_WRITE* is defined as 0 and does not define a flag; it is provided as a mnemonic convenience to callers of routines like *swap* which have a separate argument which indicates read or write.

B_DONE This bit is set to 0 when a block is handed to the the device strategy routine and is turned on when the operation completes, whether normally as the result of an error. It is also used as part of the return argument of *getblk* to indicate if 1 that the returned buffer actually contains the data in the requested block.

B_ERROR This bit may be set to 1 when *B_DONE* is set to indicate that an I/O or other error occurred. If it is set the *b_error* byte of the buffer header may contain an error code if it is non-zero. If *b_error* is 0 the nature of the error is not specified. Actually no driver at present sets *b_error;* the latter is provided for a future improvement whereby a more detailed error-reporting scheme may be implemented.

B_BUSY This bit indicates that the buffer header is not on the free list, i.e. is dedicated to someone's exclusive use. The buffer still remains attached to the list of blocks associated with its device, however. When *getblk* (or *bread,* which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears.

B_PHYS This bit is set for raw I/O transactions that need to allocate the Unibus map on an 11/70.

B_MAP This bit is set on buffers that have the Unibus map allocated, so that the *iodone* routine knows to deallocate the map.

B_WANTED
This flag is used in conjunction with the *B_BUSY* bit. Before sleeping as described just above, *getblk* sets this flag. Conversely, when the block is freed and the busy bit goes down (in *brelse)* a *wakeup* is given for the block header whenever *B_WANTED* is on. This strategem avoids the overhead of having to call *wakeup* every time a buffer is freed on the chance that someone might want it.

B_AGE This bit may be set on buffers just before releasing them; if it is on, the buffer is placed at the head of the free list, rather than at the tail. It is a performance heuristic used when the caller judges that the same block will not soon be used again.

B_ASYNC This bit is set by *bawrite* to indicate to the appropriate device driver that the buffer should be released when the write has been finished, usually at interrupt time. The difference between *bwrite* and *bawrite* is that the former starts I/O, waits until it is done, and frees the buffer.

The latter merely sets this bit and starts I/O. The bit indicates that *relse* should be called for the buffer on completion.

B_DELWRIThis bit is set by *bdwrite* before releasing the buffer. When *getblk,* while searching for a free block, discovers the bit is 1 in a buffer it would otherwise grab, it causes the block to be written out before reusing it.

Block Device Drivers

The *bdevsw* table contains the names of the interface routines and that of a table for each device.

Just as for character devices, block device drivers may supply an *open* and a *close* routine called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a *strategy* routine which is called with a pointer to a buffer header as argument. As discussed, the buffer header contains a read/write flag, the core address, the block number, a (negative) word count, and the major and minor device number. The role of the strategy routine is to carry out the operation as requested by the information in the buffer header. When the transaction is complete the *B_DONE* (and possibly the *B_ERROR)* bits should be set. Then if the *B_ASYNC* bit is set, *brelse* should be called; otherwise, *wakeup.* In cases where the device is capable, under error-free operation, of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header; otherwise, the residual count should be set to 0. This particular mechanism is really for the benefit of the magtape driver; when reading this device records shorter than requested are quite normal, and the user should be told the actual length of the record.

Although the most usual argument to the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example the *swap* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this device. Care has to be taken that no extraneous bits get turned on in the flag word.

The device's table specified by *bdevsw* has a byte to contain an active flag and an error count, a pair of links which constitute the head of the chain of buffers for the device *(b_forw, b_back),* and a first and last pointer for a device queue. Of these things, all are used solely by the device driver itself except for the buffer-chain pointers. Typically the flag encodes the state of the device, and is used at a minimum to indicate that the device is currently engaged in transferring information and no new command should be issued. The error count is useful for counting retries when errors occur. The device queue is used to remember stacked requests; in the simplest case it may be maintained as a first-in first-out list. Since buffers which have been handed over to the strategy routines are never on the list of free buffers, the pointers in the buffer which maintain the free list *(av_forw, av_back)* are also used to contain the pointers which maintain the device queues.

A couple of routines are provided which are useful to block device drivers. *iodone(bp)* arranges that the buffer to which *bp* points be released or awakened, as appropriate, when the strategy module has finished with the buffer, either normally or after an error. (In the latter case the *B_ERROR* bit has presumably been set.)

The routine *geterror(bp)* can be used to examine the error bit in a buffer header and arrange that any error indication found therein is reflected to the user. It may be called only in the non-interrupt part of a driver when I/O has completed *(B_DONE* has been set).

Raw Block-device I/O

A scheme has been set up whereby block device drivers may provide the ability to transfer information directly between the user's core image and the device without the use of buffers and in blocks as large as the caller requests. The method involves setting up a character-type special file corresponding to the raw device and providing *read* and *write* routines which set up what is usually a private, non-shared buffer header with the appropriate information and call the device's strategy routine. If desired, separate *open* and *close* routines may be provided but this is usually unnecessary. A special-function routine might come in handy, especially for magtape.

A great deal of work has to be done to generate the "appropriate information" to put in the argument buffer for the strategy module; the worst part is to map relocated user addresses to physical addresses. Most of this work is done by *physio(strat, bp, dev, rw)* whose arguments are the name of the strategy routine *strat,* the buffer pointer *bp,* the device number *dev,* and a read-write flag *rw* whose value is either *B_READ* or *B_WRITE. Physio* makes sure that the user's base address and count are even (because most devices work in words) and that the core area affected is contiguous in physical space; it delays until the buffer is not busy, and makes it busy while the operation is in progress; and it sets up user error return information.

The Programming Language EFL

Stuart I. Feldman

Fortran
Preprocessors
Ratfor

ABSTRACT

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. The EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment. EFL can be viewed as a descendant of B. W. Kernighan's Ratfor [1]; the name originally stood for 'Extended Fortran Language'. The current version of the EFL compiler is written in portable C.

1. INTRODUCTION

1.1. Purpose

EFL is a clean, general purpose computer language intended to encourage portable programming. It has a uniform and readable syntax and good data and control flow structuring. EFL programs can be translated into efficient Fortran code, so the EFL programmer can take advantage of the ubiquity of Fortran, the valuable libraries of software written in that language, and the portability that comes with the use of a standardized language, without suffering from Fortran's many failings as a language. It is especially useful for numeric programs. Thus, the EFL language permits the programmer to express complicated ideas in a comprehensible way, while permitting access to the power of the Fortran environment.

1.2. History

EFL can be viewed as a descendant of B. W. Kernighan's Ratfor [1]; the name originally stood for 'Extended Fortran Language'. A. D. Hall designed the initial version of the language and wrote a preliminary version of a compiler. I extended and modified the language and wrote a full compiler (in C) for it. The current compiler is much more than a simple preprocessor: it attempts to diagnose all syntax errors, to provide readable Fortran output, and to avoid a number of niggling restrictions. To achieve this goal, a sizable two-pass translator is needed.

1.3. Notation

In examples and syntax specifications, **boldface** type is used to indicate literal words and punctuation, such as **while**. Words in *italic* type indicate an item in a category, such as an *expression*. A construct surrounded by double brackets represents a list of one or more of those items, separated by commas. Thus, the notation *item*

could refer to any of the following:

item
item, item
item, item, item

The reader should have a fair degree of familiarity with some procedural language. There will be occasional references to Ratfor and to Fortran which may be ignored if the reader is unfamiliar with those languages.

2. LEXICAL FORM

2.1. Character Set

The following characters are legal in an EFL program:

letters	**a b c d e f g h i j k l m**
	n o p q r s t u v w x y z
digits	**0 1 2 3 4 5 6 7 8 9**
white space	*blank tab*
quotes	' "
sharp	#
continuation	_
braces	{ }
parentheses	( )
other	, ; : . + − * /
	= < > & ˜ | $

Letter case (upper or lower) is ignored except within strings, so 'a' and 'A' are treated as the same character. All of the examples below are printed in lower case. An exclamation mark ('!') may be used in place of a tilde ('˜'). Square brackets ('[' and ']') may be used in place of braces ('{' and '}').

2.2. Lines

EFL is a line-oriented language. Except in special cases (discussed below), the end of a line marks the end of a token and the end of a statement. The trailing portion of a line may be used for a comment. There is a mechanism for diverting input from one source file to another, so a single line in the program may be replaced by a number of lines from the other file. Diagnostic messages are labeled with the line number of the file on which they are detected.

2.2.1. White Space

Outside of a character string or comment, any sequence of one or more spaces or tab characters acts as a single space. Such a space terminates a token.

2.2.2. Comments

A comment may appear at the end of any line. It is introduced by a sharp (#) character, and continues to the end of the line. (A sharp inside of a quoted string does not mark a comment.) The sharp and succeeding characters on the line are discarded. A blank line is also a comment. Comments have no effect on execution.

2.2.3. Include Files

It is possible to insert the contents of a file at a point in the source text, by referencing it in a line like

```
include joe
```

No statement or comment may follow an **include** on a line. In effect, the **include** line is replaced by the lines in the named file, but diagnostics refer to the line number in the included file. **Includes** may be nested at least ten deep.

2.2.4. Continuation

Lines may be continued explicitly by using the underscore (_) character. If the last character of a line (after comments and trailing white space have been stripped) is an underscore, the end of line and the initial blanks on the next line are ignored. Underscores are ignored in other contexts (except inside of quoted strings). Thus

```
1_000_000_
   000
```

equals 10^9.

There are also rules for continuing lines automatically: the end of line is ignored whenever it is obvious that the statement is not complete. To be specific, a statement is continued if the last token on a line is an operator, comma, left brace, or left parenthesis. (A statement is not continued just because of unbalanced braces or parentheses.) Some compound statements are also continued automatically; these points are noted in the sections on executable statements.

2.2.5. Multiple Statements on a Line

A semicolon terminates the current statement. Thus, it is possible to write more than one statement on a line. A line consisting only of a semicolon, or a semicolon following a semicolon, forms a null statement.

2.3. Tokens

A program is made up of a sequence of tokens. Each token is a sequence of characters. A blank terminates any token other than a quoted string. End of line also terminates a token unless explicit continuation (see above) is signaled by an underscore.

2.3.1. Identifiers

An identifier is a letter or a letter followed by letters or digits. The following is a list of the reserved words that have special meaning in EFL. They will be discussed later.

array	**exit**	**precision**
automatic	**external**	**procedure**
break	**false**	**read**
call	**field**	**readbin**
case	**for**	**real**
character	**function**	**repeat**
common	**go**	**return**
complex	**goto**	**select**
continue	**if**	**short**
debug	**implicit**	**sizeof**
default	**include**	**static**
define	**initial**	**struct**
dimension	**integer**	**subroutine**
do	**internal**	**true**
double	**lengthof**	**until**
doubleprecision	**logical**	**value**
else	**long**	**while**
end	**next**	**write**
equivalence	**option**	**writebin**

The use of these words is discussed below. These words may not be used for any other purpose.

2.3.2. Strings

A character string is a sequence of characters surrounded by quotation marks. If the string is bounded by single-quote marks (´), it may contain double quote marks ("), and vice versa. A quoted string may not be broken across a line boundary.

´hello there´
"ain´t misbehavin´"

2.3.3. Integer Constants

An integer constant is a sequence of one or more digits.

```
0
57
123456
```

2.3.4. Floating Point Constants

A floating point constant contains a dot and/or an exponent field. An *exponent field* is a letter **d** or **e** followed by an optionally signed integer constant. If *I* and *J* are integer constants and *E* is an exponent field, then a floating constant has one of the following forms:

J
I.
IJ
IE
I.E
JE
IJE

2.3.5. Punctuation

Certain characters are used to group or separate objects in the language. These are

parentheses	( )
braces	{ }
comma	,
semicolon	;
colon	:
end-of-line	

The end-of-line is a token (statement separator) when the line is neither blank nor continued.

2.3.6. Operators

The EFL operators are written as sequences of one or more non-alphanumeric characters.

```
+  −  *  /  **
<  <=  >  >=  ==  ˜=
&&  ||  &  |
+=  −=    /=  **=
&&=  ||=    &=  |=
−>  .  $
```

A dot ('.') is an operator when it qualifies a structure element name, but not when it acts as a decimal point in a numeric constant. There is a special mode (see the Atavisms section) in which some of the operators may be represented by a string consisting of a dot, an identifier, and a dot (*e.g.*, .lt.).

2.4. Macros

EFL has a simple macro substitution facility. An identifier may be defined to be equal to a string of tokens; whenever that name appears as a token in the program, the string replaces it. A macro name is given a value in a **define** statement like

```
define count    n += 1
```

Any time the name **count** appears in the program, it is replaced by the statement

n += 1

A **define** statement must appear alone on a line; the form is

define *name rest-of-line*

Trailing comments are part of the string.

3. PROGRAM FORM

3.1. Files

A *file* is a sequence of lines. A file is compiled as a single unit. It may contain one or more procedures. Declarations and options that appear outside of a procedure affect the succeeding procedures on that file.

3.2. Procedures

Procedures are the largest grouping of statements in EFL. Each procedure has a name by which it is invoked. (The first procedure invoked during execution, known as the *main* procedure, has the null name.) Procedure calls and argument passing are discussed in Section 8.

3.3. Blocks

Statements may be formed into groups inside of a procedure. To describe the scope of names, it is convenient to introduce the ideas of *block* and of *nesting level.* The beginning of a program file is at nesting level zero. Any options, macro definitions, or variable declarations there are also at level zero. The text immediately following a **procedure** statement is at level 1. After the declarations, a left brace marks the beginning of a new block and increases the nesting level by 1; a right brace drops the level by 1. (Braces inside declarations do not mark blocks.) (See Section 7.2). An **end** statement marks the end of the procedure, level 1, and the return to level 0. A name (variable or macro) that is defined at level *k* is defined throughout that block and in all deeper nested levels in which that name is not redefined or redeclared. Thus, a procedure might look like the following:

```
#  block 0
procedure george
real x
x = 2
 . . .
if(x > 2)
          {           # new block
          integer x   # a different variable
          do x = 1,7
                      write(,x)
          . . .
          }           # end of block
end       # end of procedure, return to block 0
```

3.4. Statements

A statement is terminated by end of line or by a semicolon. Statements are of the following types:

Option
Include
Define

Procedure
End

Declarative
Executable

The **option** statement is described in Section 10. The **include**, **define**, and **end** statements have been described above; they may not be followed by another statement on a line. Each procedure begins with a **procedure** statements and finishes with an **end** statement; these are discussed in Section 8. Declarations describe types and values of variables and procedures. Executable statements cause specific actions to be taken. A block is an example of an executable statement; it is made up of declarative and executable statements.

3.5. Labels

An executable statement may have a *label* which may be used in a branch statement. A label is an identifier followed by a colon, as in

```
            read(, x)
            if(x < 3) goto error
            . . .
error:      fatal("bad input")
```

4. DATA TYPES AND VARIABLES

EFL supports a small number of basic (scalar) types. The programmer may define objects made up of variables of basic type; other aggregates may then be defined in terms of previously defined aggregates.

4.1. Basic Types

The basic types are

logical
integer
field(*m*:*n*)
real
complex
long real
long complex
character(*n*)

A logical quantity may take on the two values true and false. An integer may take on any whole number value in some machine-dependent range. A field quantity is an integer restricted to a particular closed interval ([*m*:*n*]). A 'real' quantity is a floating point approximation to a real or rational number. A long real is a more precise approximation to a rational. (Real quantities are represented as single precision floating point numbers; long reals are double precision floating point numbers.) A complex quantity is an approximation to a complex number, and is represented as a pair of reals. A character quantity is a fixed-length string of *n* characters.

4.2. Constants

There is a notation for a constant of each basic type.

A logical may take on the two values

true
false

An integer or field constant is a fixed point constant, optionally preceded by a plus or minus sign, as in

17
−94
+6
0

A long real ('double precision') constant is a floating point constant containing an exponent field that begins with the letter **d**. A real ('single precision') constant is any other floating point constant. A real or long real constant may be preceded by a plus or minus sign. The following are valid real constants:

```
17.3
-.4
7.9e-6   ( = 7.9×10^-6)
14e9   ( = 1.4×10^10)
```

The following are valid long real constants

```
7.9d-6   ( = 7.9×10^-6)
5d3
```

A character constant is a quoted string.

4.3. Variables

A variable is a quantity with a name and a location. At any particular time the variable may also have a value. (A variable is said to be *undefined* before it is initialized or assigned its first value, and after certain indefinite operations are performed.) Each variable has certain attributes:

4.3.1. Storage Class

The association of a name and a location is either transitory or permanent. Transitory association is achieved when arguments are passed to procedures. Other associations are permanent (static). (A future extension of EFL may include dynamically allocated variables.)

4.3.2. Scope of Names

The names of common areas are global, as are procedure names: these names may be used anywhere in the program. All other names are local to the block in which they are declared.

4.3.3. Precision

Floating point variables are either of normal or **long** precision. This attribute may be stated independently of the basic type.

4.4. Arrays

It is possible to declare rectangular arrays (of any dimension) of values of the same type. The index set is always a cross-product of intervals of integers. The lower and upper bounds of the intervals must be constants for arrays that are local or **common.** A formal argument array may have intervals that are of length equal to one of the other formal arguments. An element of an array is denoted by the array name followed by a parenthesized comma-separated list of integer values, each of which must lie within the corresponding interval. (The intervals may include negative numbers.) Entire arrays may be passed as procedure arguments or in input/output lists, or they may be initialized; all other array references must be to individual elements.

4.5. Structures

It is possible to define new types which are made up of elements of other types. The compound object is known as a *structure;* its constituents are called *members* of the structure. The structure may be given a name, which acts as a type name in the remaining statements within the scope of its declaration. The elements of a structure may be of any type (including previously defined structures), or they may be arrays of such objects. Entire structures may be passed to procedures or be used in input/output lists; individual elements of structures may be referenced. The uses of structures will be detailed below. The following structure might represent a symbol table:

```
struct tableentry
        {
        character(8) name
        integer hashvalue
        integer numberofelements
        field(0:1) initialized, used, set
        field(0:10) type
        }
```

5. EXPRESSIONS

Expressions are syntactic forms that yield a value. An expression may have any of the following forms, recursively applied:

primary
(*expression*)
unary-operator expression
expression binary-operator expression

In the following table of operators, all operators on a line have equal precedence and have higher precedence than operators on later lines. The meanings of these operators are described in sections 5.3 and 5.4.

```
-> .
**
* /    unary + - ++ --
+ -
< <= > >= == ~=
& &&
| ||
$
= += -= *= /= **=  &= |= &&= ||=
```

Examples of expressions are

```
a<b && b<c
-(a + sin(x)) / (5+cos(x))**2
```

5.1. Primaries

Primaries are the basic elements of expressions, as follows:

5.1.1. Constants

Constants are described in Section 4.2.

5.1.2. Variables

Scalar variable names are primaries. They may appear on the left or the right side of an assignment. Unqualified names of aggregates (structures or arrays) may only appear as procedure arguments and in input/output lists.

5.1.3. Array Elements

An element of an array is denoted by the array name followed by a parenthesized list of subscripts, one integer value for each declared dimension:

```
a(5)
b(6,-3,4)
```

5.1.4. Structure Members

A structure name followed by a dot followed by the name of a member of that structure constitutes a reference to that element. If that element is itself a structure, the reference may be further qualified.

```
a.b
x(3).y(4).z(5)
```

5.1.5. Procedure Invocations

A procedure is invoked by an expression of one of the forms

procedurename ( )
procedurename (*expression*)
procedurename (*expression-1*, ..., *expression-n*)

The *procedurename* is either the name of a variable declared **external** or it is the name of a function known to the EFL compiler (see Section 8.5), or it is the actual name of a procedure, as it appears in a **procedure** statement. If a *procedurename* is declared **external** and is an argument of the current procedure, it is associated with the procedure name passed as actual argument; otherwise it is the actual name of a procedure. Each *expression* in the above is called an *actual argument*. Examples of procedure invocations are

```
f(x)
work()
g(x, y+3, 'xx')
```

When one of these procedure invocations is to be performed, each of the actual argument expressions is first evaluated. The types, precisions, and bounds of actual and formal arguments should agree. If an actual argument is a variable name, array element, or structure member, the called procedure is permitted to use the corresponding formal argument as the left side of an assignment or in an input list; otherwise it may only use the value. After the formal and actual arguments are associated, control is passed to the first executable statement of the procedure. When a **return** statement is executed in that procedure, or when control reaches the **end** statement of that procedure, the function value is made available as the value of the procedure invocation. The type of the value is determined by the attributes of the *procedurename* that are declared or implied in the calling procedure, which must agree with the attributes declared for the function in its procedure. In the special case of a generic function, the type of the result is also affected by the type of the argument. See Chapter 8 for details.

5.1.6. Input/Output Expressions

The EFL input/output syntactic forms may be used as integer primaries that have a non-zero value if an error occurs during the input or output. See Section 7.7.

5.1.7. Coercions

An expression of one precision or type may be converted to another by an expression of the form

attributes (*expression*)

At present, the only *attributes* permitted are precision and basic types. Attributes are separated by white space. An arithmetic value of one type may be coerced to any other arithmetic type; a character expression of one length may be coerced to a character expression of another length; logical expressions may not be coerced to a nonlogical type. As a special case, a quantity of **complex** or **long complex** type may be constructed from two integer or real quantities by passing two expressions (separated by a comma) in the coercion. Examples and equivalent values are

integer(5.3) = 5
long real(5) = 5.0d0
complex(5,3) = 5+3*i*

Most conversions are done implicitly, since most binary operators permit operands of different arithmetic types. Explicit coercions are of most use when it is necessary to convert the type of an actual argument to match that of the corresponding formal parameter in a procedure call.

5.1.8. Sizes

There is a notation which yields the amount of memory required to store a datum or an item of specified type:

sizeof (*leftside*)
sizeof (*attributes*)

In the first case, *leftside* can denote a variable, array, array element, or structure member. The value of **sizeof** is an integer, which gives the size in arbitrary units. If the size is needed in terms of the size of some specific unit, this can be computed by division:

sizeof(x) / sizeof(integer)

yields the size of the variable **x** in integer words.

The distance between consecutive elements of an array may not equal **sizeof** because certain data types require final padding on some machines. The **lengthof** operator gives this larger value, again in arbitrary units. The syntax is

lengthof (*leftside*)
lengthof (*attributes*)

5.2. Parentheses

An expression surrounded by parentheses is itself an expression. A parenthesized expression must be evaluated before an expression of which it is a part is evaluated.

5.3. Unary Operators

All of the unary operators in EFL are prefix operators. The result of a unary operator has the same type as its operand.

5.3.1. Arithmetic

Unary + has no effect. A unary – yields the negative of its operand.

The prefix operator ++ adds one to its operand. The prefix operator — subtracts one from its operand. The value of either expression is the result of the addition or subtraction. For these two operators, the operand must be a scalar, array element, or structure member of arithmetic type. (As a side effect, the operand value is changed.)

5.3.2. Logical

The only logical unary operator is complement (˜). This operator is defined by the equations

˜ true = false
˜ false = true

5.4. Binary Operators

Most EFL operators have two operands, separated by the operator. Because the character set must be limited, some of the operators are denoted by strings of two or three special characters. All binary operators except exponentiation are left associative.

5.4.1. Arithmetic

The binary arithmetic operators are

+	addition
−	subtraction
*	multiplication
/	division
**	exponentiation

Exponentiation is right associative: a**b**c = a**(b**c) = $a^{(b^c)}$ The operations have the conventional meanings: 8+2 = 10, 8−2 = 6, 8*2 = 16, 8/2 = 4, 8**2 = 8^2 = 64.

The type of the result of a binary operation *A op B* is determined by the types of its operands:

	Type of B				
Type of A	integer	real	long real	complex	long complex
integer	integer	real	long real	complex	long complex
real	real	real	long real	complex	long complex
long real	long real	long real	long real	long complex	long complex
complex	complex	complex	long complex	complex	long complex
long complex	long complex	long complex	long complex	long complex	long complex

If the type of an operand differs from the type of the result, the calculation is done as if the operand were first coerced to the type of the result. If both operands are integers, the result is of type integer, and is computed exactly. (Quotients are truncated toward zero, so 8/3=2.)

5.4.2. Logical

The two binary logical operations in EFL, **and** and **or**, are defined by the truth tables:

A	B	A and B	A or B
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Each of these operators comes in two forms. In one form, the order of evaluation is specified. The expression

a && b

is evaluated by first evaluating **a**; if it is false then the expression is false and **b** is not evaluated; otherwise the expression has the value of **b**. The expression

a || b

is evaluated by first evaluating **a**; if it is true then the expression is true and **b** is not evaluated; otherwise the expression has the value of **b**. The other forms of the operators (& for **and** and | for **or**) do not imply an order of evaluation. With the latter operators, the compiler may speed up the code by evaluating the operands in any order.

5.4.3. Relational Operators

There are six relations between arithmetic quantities. These operators are not associative.

EFL Operator	Meaning	
<	<	less than
<=	≤	less than or equal to
==	=	equal to
~=	≠	not equal to
>	>	greater than
>=	≥	greater than or equal

Since the complex numbers are not ordered, the only relational operators that may take complex operands are == and ~= . The character collating sequence is not defined.

5.4.4. Assignment Operators

All of the assignment operators are right associative. The simple form of assignment is

basic-left-side = *expression*

A *basic-left-side* is a scalar variable name, array element, or structure member of basic type. This statement computes the expression on the right side, and stores that value (possibly after coercing the value to the type of the left side) in the location named by the left side. The value of the assignment expression is the value assigned to the left side after coercion.

There is also an assignment operator corresponding to each binary arithmetic and logical operator. In each case, *a op= b* is equivalent to *a = a op b*. (The operator and equal sign must not be separated by blanks.) Thus, **n+=2** adds 2 to n. The location of the left side is evaluated only once.

5.5. Dynamic Structures

EFL does not have an address (pointer, reference) type. However, there is a notation for dynamic structures,

leftside –> *structurename*

This expression is a structure with the shape implied by *structurename* but starting at the location of *leftside*. In effect, this overlays the structure template at the specified location. The *leftside* must be a variable, array, array element, or structure member. The type of the *leftside* must be one of the types in the structure declaration. An element of such a structure is denoted in the usual way using the dot operator. Thus,

place(i) –> st.elt

refers to the **elt** member of the **st** structure starting at the i^{th} element of the array **place**.

5.6. Repetition Operator

Inside of a list, an element of the form

integer-constant-expression **$** *constant-expression*

is equivalent to the appearance of the *expression* a number of times equal to the first expression. Thus,

(3, 3$4, 5)

is equivalent to

(3, 4, 4, 4, 5)

5.7. Constant Expressions

If an expression is built up out of operators (other than functions) and constants, the value of the expression is a constant, and may be used anywhere a constant is required.

6. DECLARATIONS

Declarations statement describe the meaning, shape, and size of named objects in the EFL language.

6.1. Syntax

A declaration statement is made up of attributes and variables. Declaration statements are of two form:

attributes variable-list
attributes { declarations }

In the first case, each name in the *variable-list* has the specified attributes. In the second, each name in the declarations also has the specified attributes. A variable name may appear in more than one variable list, so long as the attributes are not contradictory. Each name of a nonargument variable may be accompanied by an initial value specification. The *declarations* inside the braces are one or more declaration statements. Examples of declarations are

```
integer k=2

long real b(7,3)

common(cname)
    {
    integer i
    long real array(5,0:3) x, y
    character(7) ch
    }
```

6.2. Attributes

6.2.1. Basic Types

The following are basic types in declarations

logical
integer
field(m:n)
character(k)
real
complex

In the above, the quantities k, m, and n denote integer constant expressions with the properties $k>0$ and $n>m$.

6.2.2. Arrays

The dimensionality may be declared by an array attribute

array(b_1,...,b_n)

Each of the b_i may either be a single integer expression or a pair of integer expressions separated by a colon. The pair of expressions form a lower and an upper bound; the single expression is an upper bound with an implied lower bound of 1. The number of dimensions is equal to n, the number of bounds. All of the integer expressions must be constants. An exception is permitted only if all of the variables associated with an array declarator are formal arguments of the procedure; in this case, each bound must have the property that *upper*$-$*lower*$+1$ is equal to a formal argument of the procedure. (The compiler has limited ability to simplify expressions, but it will recognize important cases such as (0:n−1). The upper bound for the last dimension (b_n) may be marked by an asterisk (*) if the size of the array is not known. The following are legal **array** attributes:

```
array(5)
array(5, 1:5, -3:0)
array(5, *)
array(0:m-1, m)
```

6.2.3. Structures

A structure declaration is of the form

struct *structname* { *declaration statements* }

The *structname* is optional; if it is present, it acts as if it were the name of a type in the rest of its scope. Each name that appears inside the *declarations* is a *member* of the structure, and has a special meaning when used to qualify any variable declared with the structure type. A name may appear as a member of any number of structures, and may also be the name of an ordinary variable, since a structure member name is used only in contexts where the parent type is known. The following are valid structure attributes

```
struct xx
    {
    integer a, b
    real x(5)
    }

struct { xx z(3); character(5) y }
```

The last line defines a structure containing an array of three **xx**'*s* and a character string.

6.2.4. Precision

Variables of floating point (**real** or **complex**) type may be declared to be **long** to ensure they have higher precision than ordinary floating point variables. The default precision is **short**.

6.2.5. Common

Certain objects called *common areas* have external scope, and may be referenced by any procedure that has a declaration for the name using a

common (*commonareaname*)

attribute. All of the variables declared with a particular **common** attribute are in the same block; the order in which they are declared is significant. Declarations for the same block in differing procedures must have the variables in the same order and with the same types, precision, and shapes, though not necessarily with the same names.

6.2.6. External

If a name is used as the procedure name in a procedure invocation, it is implicitly declared to have the **external** attribute. If a procedure name is to be passed as an argument, it is necessary to declare it in a statement of the form

external *name*

If a name has the external attribute and it is a formal argument of the procedure, then it is associated with a procedure identifier passed as an actual argument at each call. If the name is not a formal argument, then that name is the actual name of a procedure, as it appears in the corresponding procedure statement.

6.3. Variable List

The elements of a variable list in a declaration consist of a name, an optional dimension specification, and an optional initial value specification. The name follows the usual rules. The dimension specification is the same form and meaning as the parenthesized list in an array attribute. The initial value specification is an equal sign (=) followed by a constant expression. If the name is an array, the right side of the equal sign may be a parenthesized list of constant expressions, or repeated elements or lists; the total number of elements in the list must not exceed the number of elements of the array, which are filled in column-major order.

6.4. The Initial Statement

An initial value may also be specified for a simple variable, array, array element, or member of a structure using a statement of the form

initial *var* = *val*

The *var* may be a variable name, array element specification, or member of structure. The right side follows the same rules as for an initial value specification in other declaration statements.

7. EXECUTABLE STATEMENTS

Every useful EFL program contains executable statements — otherwise it would not do anything and would not need to be run. Statements are frequently made up of other statements. Blocks are the most obvious case, but many other forms contain statements as constituents.

To increase the legibility of EFL programs, some of the statement forms can be broken without an explicit continuation. A square (□) in the syntax represents a point where the end of a line will be ignored.

7.1. Expression Statements

7.1.1. Subroutine Call

A procedure invocation that returns no value is known as a subroutine call. Such an invocation is a statement. Examples are

```
work(in, out)
run()
```

Input/output statements (see Section 7.7) resemble procedure invocations but do not yield a value. If an error occurs the program stops.

7.1.2. Assignment Statements

An expression that is a simple assignment (=) or a compound assignment (+= etc.) is a statement:

```
a = b
a = sin(x)/6
x *= y
```

7.2. Blocks

A block is a compound statement that acts as a statement. A block begins with a left brace, optionally followed by declarations, optionally followed by executable statements, followed by a right brace. A block may be used anywhere a statement is permitted. A block is not an expression and does not have a value. An example of a block is

```
{
integer i    # this variable is unknown outside the braces

big = 0
do i = 1,n
   if(big < a(i))
        big = a(i)
}
```

7.3. Test Statements

Test statements permit execution of certain statements conditional on the truth of a predicate.

7.3.1. If Statement

The simplest of the test statements is the **if** statement, of form

if (*logical-expression*) □ *statement*

The logical expression is evaluated; if it is true, then the *statement* is executed.

7.3.2. If-Else

A more general statement is of the form

if (*logical-expression*) □ *statement-1* □ **else** □ *statement-2*

If the expression is **true** then *statement-1* is executed, otherwise *statement-2* is executed. Either of the consequent statements may itself be an **if-else** so a completely nested test sequence is possible:

```
if(x<y)
   if(a<b)
      k = 1
   else
      k = 2
else
   if(a<b)
      m = 1
   else
      m = 2
```

An **else** applies to the nearest preceding un-elsed **if**. A more common use is as a sequential test:

```
if(x==1)
   k = 1
else if(x==3 | x==5)
   k = 2
else
   k = 3
```

7.3.3. Select Statement

A multiway test on the value of a quantity is succinctly stated as a **select** statement, which has the general form

select(*expression*) □ *block*

Inside the block two special types of labels are recognized. A prefix of the form

case *constant* **:**

marks the statement to which control is passed if the expression in the select has a value equal to one of the case constants. If the expression equals none of these constants, but there is a label **default** inside the select, a branch is taken to that point; otherwise the statement following the right brace is executed. Once execution begins at a **case** or **default** label, it continues until the next **case** or **default** is encountered. The else-if example above is better written as

```
select(x)
  {
  case 1:
    k = 1
  case 3,5:
    k = 2
  default:
    k = 3
  }
```

Note that control does not 'fall through' to the next case.

7.4. Loops

The loop forms provide the best way of repeating a statement or sequence of operations. The simplest (**while**) form is theoretically sufficient, but it is very convenient to have the more general loops available, since each expresses a mode of control that arises frequently in practice.

7.4.1. While Statement

This construct has the form

while (*logical-expression*) □ *statement*

The expression is evaluated; if it is true, the statement is executed, and then the test is performed again. If the expression is false, execution proceeds to the next statement.

7.5. For Statement

The **for** statement is a more elaborate looping construct. It has the form

for (*initial-statement* , □ *logical-expression* , □ *iteration-statement*) □ *body-statement*

Except for the behavior of the **next** statement (see Section 7.6.3), this construct is equivalent to

```
initial-statement
while ( logical-expression )
  {
  body-statement
  iteration-statement
  }
```

This form is useful for general arithmetic iterations, and for various pointer-type operations. The sum of the integers from 1 to 100 can be computed by the fragment

```
n = 0
for(i = 1, i <= 100, i += 1)
    n += i
```

Alternatively, the computation could be done by the single statement

```
for( { n = 0 ; i = 1 } , i<=100 , { n += i ; ++i } )
    ;
```

Note that the body of the **for** loop is a null statement in this case. An example of following a linked list will be given later.

7.5.1. Repeat Statement

The statement

repeat □ *statement*

executes the statement, then does it again, without any termination test. Obviously, a test inside the *statement* is needed to stop the loop.

7.5.2. Repeat...Until Statement

The **while** loop performs a test before each iteration. The statement

repeat □ *statement* □ **until** (*logical-expression*)

executes the *statement*, then evaluates the logical; if the logical is true the loop is complete; otherwise control returns to the *statement*. Thus, the body is always executed at least once. The **until** refers to the nearest preceding **repeat** that has not been paired with an **until**. In practice, this appears to be the least frequently used looping construct.

7.5.3. Do Loops

The simple arithmetic progression is a very common one in numerical applications. EFL has a special loop form for ranging over an ascending arithmetic sequence

do *variable = expression-1, expression-2, expression-3*
  statement

The variable is first given the value *expression-1*. The statement is executed, then *expression-3* is added to the variable. The loop is repeated until the variable exceeds *expression-2*. If *expression-3* and the preceding comma are omitted, the increment is taken to be 1. The loop above is equivalent to

```
t2 = expression-2
t3 = expression-3
for(variable = expression-1 , variable <= t2 , variable += t3)
   statement
```

(The compiler translates EFL **do** statements into Fortran DO statements, which are in turn usually compiled into excellent code.) The **do** *variable* may not be changed inside of the loop, and *expression-1* must not exceed *expression-2*. The sum of the first hundred positive integers could be computed by

```
n = 0
do i = 1, 100
       n += i
```

7.6. Branch Statements

Most of the need for branch statements in programs can be averted by using the loop and test constructs, but there are programs where they are very useful.

7.6.1. Goto Statement

The most general, and most dangerous, branching statement is the simple unconditional

goto *label*

After executing this statement, the next statement performed is the one following the given label. Inside of a select the case labels of that block may be used as labels, as in the following example:

```
select(k)
        {
        case 1:
                error(7)

        case 2:
                k = 2
                goto case 4

        case 3:
                k = 5
                goto case 4

        case 4:
                fixup(k)
                goto default

        default:
                prmsg("ouch")
        }
```

(If two **select** statements are nested, the case labels of the outer select are not accessible from the inner one.)

7.6.2. Break Statement

A safer statement is one which transfers control to the statement following the current **select** or loop form. A statement of this sort is almost always needed in a **repeat** loop:

```
repeat
  {
  do a computation
  if( finished )
      break
  }
```

More general forms permit controlling a branch out of more than one construct.

```
break 3
```

transfers control to the statement following the third loop and/or select surrounding the statement. It is possible to specify which type of construct (**for, while, repeat, do,** or **select**) is to be counted. The statement

```
break while
```

breaks out of the first surrounding **while** statement. Either of the statements

```
break 3 for
break for 3
```

will transfer to the statement after the third enclosing **for** loop.

7.6.3. Next Statement

The **next** statement causes the first surrounding loop statement to go on to the next iteration: the next operation performed is the test of a **while**, the *iteration-statement* of a **for**, the body of a **repeat**, the test of a **repeat...until**, or the increment of a **do**. Elaborations similar to those for **break** are available:

next
next 3
next 3 for
next for 3

A **next** statement ignores **select** statements.

7.6.4. Return

The last statement of a procedure is followed by a return of control to the caller. If it is desired to effect such a return from any other point in the procedure, a

return

statement may be executed. Inside a function procedure, the function value is specified as an argument of the statement:

return (*expression*)

7.7. Input/Output Statements

EFL has two input statements (**read** and **readbin**), two output statements (**write** and **writebin**), and three control statements (**endfile**, **rewind**, and **backspace**). These forms may be used either as a primary with a **integer** value or as a statement. If an exception occurs when one of these forms is used as a statement, the result is undefined but will probably be treated as a fatal error. If they are used in a context where they return a value, they return zero if no exception occurs. For the input forms, a negative value indicates end-of-file and a positive value an error. The input/output part of EFL very strongly reflects the facilities of Fortran.

7.7.1. Input/Output Units

Each I/O statement refers to a 'unit', identified by a small positive integer. Two special units are defined by EFL, the *standard input unit* and the *standard output unit*. These particular units are assumed if no unit is specified in an I/O transmission statement.

The data on the unit are organized into *records*. These records may be read or written in a fixed sequence, and each transmission moves an integral number of records. Transmission proceeds from the first record until the *end of file*.

7.7.2. Binary Input/Output

The **readbin** and **writebin** statements transmit data in a machine-dependent but swift manner. The statements are of the form

writebin(*unit* , *binary-output-list*)
readbin(*unit* , *binary-input-list*)

Each statement moves one unformatted record between storage and the device. The *unit* is an integer expression. A *binary-output-list* is an *iolist* (see below) without any format specifiers. A *binary-input-list* is an *iolist* without format specifiers in which each of the expressions is a variable name, array element, or structure member.

7.7.3. Formatted Input/Output

The **read** and **write** statements transmit data in the form of lines of characters. Each statement moves one or more records (lines). Numbers are translated into decimal notation. The exact form of the lines is determined by format specifications, whether provided explicitly in the statement or implicitly. The syntax of the statements is

write(*unit* **,** *formatted-output-list* **)**
read(*unit* **,** *formatted-input-list* **)**

The lists are of the same form as for binary I/O, except that the lists may include format specifications. If the *unit* is omitted, the standard input or output unit is used.

7.7.4. Iolists

An *iolist* specifies a set of values to be written or a set of variables into which values are to be read. An *iolist* is a list of one or more *ioexpressions* of the form

expression
{ *iolist* }
do-specification { *iolist* }

For formatted I/O, an *ioexpression* may also have the forms

ioexpression : *format-specifier*
: *format-specifier*

A *do-specification* looks just like a **do** statement, and has a similar effect: the values in the braces are transmitted repeatedly until the **do** execution is complete.

7.7.5. Formats

The following are permissible *format-specifiers*. The quantities *w*, *d*, and *k* must be integer constant expressions.

i(*w***)**	integer with *w* digits
f(*w*,*d***)**	floating point number of *w* characters, *d* of them to the right of the decimal point.
e(*w*,*d***)**	floating point number of *w* characters, *d* of them to the right of the decimal point, with the exponent field marked with the letter e
l(*w***)**	logical field of width *w* characters, the first of which is **t** or **f** (the rest are blank on output, ignored on input) standing for **true** and **false** respectively
c	character string of width equal to the length of the datum
c(*w***)**	character string of width *w*
s(*k***)**	skip *k* lines
x(*k***)**	skip *k* spaces
" ... "	use the characters inside the string as a Fortran format

If no format is specified for an item in a formatted input/output statement, a default form is chosen.

If an item in a list is an array name, then the entire array is transmitted as a sequence of elements, each with its own format. The elements are transmitted in column-major order, the same order used for array initializations.

7.7.6. Manipulation statements

The three input/output statements

backspace(*unit***)**
rewind(*unit***)**
endfile(*unit***)**

look like ordinary procedure calls, but may be used either as statements or as integer expressions which yield non-zero if an error is detected. **backspace** causes the specified unit to back up, so that the next read will re-read the previous record, and the next write will over-write it. **rewind** moves the device to its beginning, so that the next input statement will read the first record. **endfile** causes the file to be marked so that the record most recently written will be the last record on the file, and any attempt to read past is an error.

8. PROCEDURES

Procedures are the basic unit of an EFL program, and provide the means of segmenting a program into separately compilable and named parts.

8.1. Procedure Statement

Each procedure begins with a statement of one of the forms

> **procedure**
> *attributes* **procedure** *procedurename*
> *attributes* **procedure** *procedurename* ( )
> *attributes* **procedure** *procedurename* (*name*)

The first case specifies the main procedure, where execution begins. In the two other cases, the *attributes* may specify precision and type, or they may be omitted entirely. The precision and type of the procedure may be declared in an ordinary declaration statement. If no type is declared, then the procedure is called a *subroutine* and no value may be returned for it. Otherwise, the procedure is a function and a value of the declared type is returned for each call. Each *name* inside the parentheses in the last form above is called a *formal argument* of the procedure.

8.2. End Statement

Each procedure terminates with a statement

end

8.3. Argument Association

When a procedure is invoked, the actual arguments are evaluated. If an actual argument is the name of a variable, an array element, or a structure member, that entity becomes associated with the formal argument, and the procedure may reference the values in the object, and assign to it. Otherwise, the value of the actual is associated with the formal argument, but the procedure may not attempt to change the value of that formal argument.

If the value of one of the arguments is changed in the procedure, it is not permitted that the corresponding actual argument be associated with another formal argument or with a **common** element that is referenced in the procedure.

8.4. Execution and Return Values

After actual and formal arguments have been associated, control passes to the first executable statement of the procedure. Control returns to the invoker either when the **end** statement of the procedure is reached or when a **return** statement is executed. If the procedure is a function (has a declared type), and a **return**(*value*) is executed, the value is coerced to the correct type and precision and returned.

8.5. Known Functions

A number of functions are known to EFL, and need not be declared. The compiler knows the types of these functions. Some of them are *generic;* i.e., they name a family of functions that differ in the types of their arguments and return values. The compiler chooses which element of the set to invoke based upon the attributes of the actual arguments.

8.5.1. Minimum and Maximum Functions

The generic functions are **min** and **max**. The **min** calls return the value of their smallest argument; the **max** calls return the value of their largest argument. These are the only functions that may take different numbers of arguments in different calls. If any of the arguments are **long real** then the result is **long real**. Otherwise, if any of the arguments are **real** then the result is **real**; otherwise all the arguments and the result must be **integer**. Examples are

```
min(5, x, -3.20)
max(i, z)
```

8.5.2. Absolute Value

The **abs** function is a generic function that returns the magnitude of its argument. For integer and real arguments the type of the result is identical to the type of the argument; for complex arguments the type of the result is the real of the same precision.

8.5.3. Elementary Functions

The following generic functions take arguments of **real**, **long real**, or **complex** type and return a result of the same type:

sin	**sine function**
cos	**cosine function**
exp	**exponential function (e^x).**
log	**natural (base e) logarithm**
log10	**common (base 10) logarithm**
sqrt	**square root function ($\sqrt{x}$).**

In addition, the following functions accept only real or long real arguments:

atan	$atan(x)=\tan^{-1}x$
atan2	$atan2(x,y)=\tan^{-1}\frac{x}{y}$

8.5.4. Other Generic Functions

The **sign** functions takes two arguments of identical type; **sign**$(x,y) = sgn(y)|x|$. The **mod** function yields the remainder of its first argument when divided by its second. These functions accept integer and real arguments.

9. ATAVISMS

Certain facilities are included in the EFL language to ease the conversion of old Fortran or Ratfor programs to EFL.

9.1. Escape Lines

In order to make use of nonstandard features of the local Fortran compiler, it is occasionally necessary to pass a particular line through to the EFL compiler output. A line that begins with a percent sign ('%') is copied through to the output, with the percent sign removed but no other change. Inside of a procedure, each escape line is treated as an executable statement. If a sequence of lines constitute a continued Fortran statement, they should be enclosed in braces.

9.2. Call Statement

A subroutine call may be preceded by the keyword **call**.

call joe
call work(17)

9.3. Obsolete Keywords

The following keywords are recognized as synonyms of EFL keywords:

Fortran	EFL
double precision	**long real**
function	**procedure**
subroutine	**procedure** *(untyped)*

9.4. Numeric Labels

Standard statement labels are identifiers. A numeric (positive integer constant) label is also permitted; the colon is optional following a numeric label.

9.5. Implicit Declarations

If a name is used but does not appear in a declaration, the EFL compiler gives a warning and assumes a declaration for it. If it is used in the context of a procedure invocation, it is assumed to be a procedure name; otherwise it is assumed to be a local variable defined at nesting level 1 in the current procedure. The assumed type is determined by the first letter of the name. The association of letters and types may be given in an **implicit** statement, with syntax

implicit (*letter-list* **)** *type*

where a *letter-list* is a list of individual letters or ranges (pair of letters separated by a minus sign). If no **implicit** statement appears, the following rules are assumed:

implicit (a–h, o–z) real
implicit (i–n) integer

9.6. Computed goto

Fortran contains an indexed multi-way branch; this facility may be used in EFL by the computed GOTO:

goto (*label* **),** *expression*

The expression must be of type integer and be positive but be no larger than the number of labels in the list. Control is passed to the statement marked by the label whose position in the list is equal to the expression.

9.7. Go To Statement

In unconditional and computed goto statements, it is permissible to separate the go and to words, as in

go to xyz

9.8. Dot Names

Fortran uses a restricted character set, and represents certain operators by multi-character sequences. There is an option (**dots=on**; see Section 10.2) which forces the compiler to recognize the forms in the second column below:

<	**.lt.**
<=	**.le.**
>	**.gt.**
>=	**.ge.**
==	**.eq.**
~=	**.ne.**
&	**.and.**
|	**.or.**
&&	**.andand.**
||	**.oror.**
~	**.not.**
true	**.true.**
false	**.false.**

In this mode, no structure element may be named **lt**, **le**, etc. The readable forms in the left column are always recognized.

9.9. Complex Constants

A complex constant may be written as a parenthesized list of real quantities, such as

(1.5, 3.0)

The preferred notation is by a type coercion,

complex(1.5, 3.0)

9.10. Function Values

The preferred way to return a value from a function in EFL is the **return(***value***)** construct. However, the name of the function acts as a variable to which values may be assigned; an ordinary **return** statement returns the last value assigned to that name as the function value.

9.11. Equivalence

A statement of the form

equivalence $v_1, v_2, \ldots, v_n$

declares that each of the v_i starts at the same memory location. Each of the v_i may be a variable name, array element name, or structure member.

9.12. Minimum and Maximum Functions

There are a number of non-generic functions in this category, which differ in the required types of the arguments and the type of the return value. They may also have variable numbers of arguments, but all the arguments must have the same type.

Function	Argument Type	Result Type
amin0	**integer**	**real**
amin1	**real**	**real**
min0	**integer**	**integer**
min1	**real**	**integer**
dmin1	**long real**	**long real**
amax0	**integer**	**real**
amax1	**real**	**real**
max0	**integer**	**integer**
max1	**real**	**integer**
dmax1	**long real**	**long real**

10. COMPILER OPTIONS

A number of options can be used to control the output and to tailor it for various compilers and systems. The defaults chosen are conservative, but it is sometimes necessary to change the output to match peculiarities of the target environment.

Options are set with statements of the form

option *opt*

where each *opt* is of one of the forms

optionname
optionname = optionvalue

The *optionvalue* is either a constant (numeric or string) or a name associated with that option. The two names yes and **no** apply to a number of options.

10.1. Default Options

Each option has a default setting. It is possible to change the whole set of defaults to those appropriate for a particular environment by using the **system** option. At present, the only valid values are **system=unix** and **system=gcos.**

10.2. Input Language Options

The **dots** option determines whether the compiler recognizes **.lt.** and similar forms. The default setting is **no.**

10.3. Input/Output Error Handling

The **ioerror** option can be given three values: **none** means that none of the I/O statements may be used in expressions, since there is no way to detect errors. The implementation of the **ibm** form uses ERR= and END= clauses. The implementation of the **fortran77** form uses IOSTAT= clauses.

10.4. Continuation Conventions

By default, continued Fortran statements are indicated by a character in column 6 (Standard Fortran). The option **continue=column1** puts an ampersand (&) in the first column of the continued lines instead.

10.5. Default Formats

If no format is specified for a datum in an iolist for a **read** or **write** statement, a default is provided. The default formats can be changed by setting certain options

Option	Type
iformat	integer
rformat	real
dformat	long real
zformat	complex
zdformat	long complex
lformat	logical

The associated value must be a Fortran format, such as

option rformat=f22.6

10.6. Alignments and Sizes

In order to implement **character** variables, structures, and the **sizeof** and **lengthof** operators, it is necessary to know how much space various Fortran data types require, and what boundary alignment properties they demand. The relevant options are

Fortran Type	Size Option	Alignment Option
integer	**isize**	**ialign**
real	**rsize**	**ralign**
long real	**dsize**	**dalign**
complex	**zsize**	**zalign**
logical	**lsize**	**lalign**

The sizes are given in terms of an arbitrary unit; the alignment is given in the same units. The option **charperint** gives the number of characters per **integer** variable.

10.7. Default Input/Output Units

The options **ftnin** and **ftnout** are the numbers of the standard input and output units. The default values are **ftnin=5** and **ftnout=6.**

10.8. Miscellaneous Output Control Options

Each Fortran procedure generated by the compiler will be preceded by the value of the **procheader** option.

No Hollerith strings will be passed as subroutine arguments if **hollincall=no** is specified.

The Fortran statement numbers normally start at 1 and increase by 1. It is possible to change the increment value by using the **deltastno** option.

11. EXAMPLES

In order to show the flavor or programming in EFL, we present a few examples. They are short, but show some of the convenience of the language.

11.1. File Copying

The following short program copies the standard input to the standard output, provided that the input is a formatted file containing lines no longer than a hundred characters.

```
procedure   # main program
character(100) line

while( read( , line) == 0 )
      write( , line)
end
```

Since **read** returns zero until the end of file (or a read error), this program keeps reading and writing until the input is exhausted.

11.2. Matrix Multiplication

The following procedure multiplies the $m \times n$ matrix a by the $n \times p$ matrix b to give the $m \times p$ matrix c. The calculation obeys the formula $c_{ij} = \sum a_{ik} b_{kj}$.

```
procedure matmul(a,b,c, m,n,p)
integer i, j, k, m, n, p
long real a(m,n), b(n,p), c(m,p)
do i = 1,m
do j = 1,p
            {
            c(i,j) = 0
            do k = 1,n
                        c(i,j) += a(i,k) * b(k,j)
            }
end
```

11.3. Searching a Linked List

Assume we have a list of pairs of numbers (x,y). The list is stored as a linked list sorted in ascending order of x values. The following procedure searches this list for a particular value of x and returns the corresponding y value.

```
define LAST          0
define NOTFOUND     -1

integer procedure val(list, first, x)

#  list is an array of structures.
#  Each structure contains a thread index value, an x, and a y value.
struct
            {
            integer nextindex
            integer x, y
            } list(*)
integer first, p, arg

for(p = first , p~=LAST && list(p).x<=x , p = list(p).nextindex)
            if(list(p).x == x)
                        return( list(p).y )

return(NOTFOUND)
end
```

The search is a single for loop that begins with the head of the list and examines items until either the list is exhausted (p==LAST) or until it is known that the specified value is not on the list (list(p).x > x). The two tests in the conjunction must be performed in the specified order to avoid using an invalid subscript in the **list(p)** reference. Therefore, the **&&** operator is used. The next element in the chain is found by the iteration statement **p=list(p).nextindex**.

11.4. Walking a Tree

As an example of a more complicated problem, let us imagine we have an expression tree stored in a common area, and that we want to print out an infix form of the tree. Each node is either a leaf (containing a numeric value) or it is a binary operator, pointing to a left and a right descendant. In a recursive language, such a tree walk would be implement by the following simple pseudocode:

```
if this node is a leaf
        print its value
otherwise
        print a left parenthesis
        print the left node
        print the operator
        print the right node
        print a right parenthesis
```

In a nonrecursive language like EFL, it is necessary to maintain an explicit stack to keep track of the current state of the computation. The following procedure calls a procedure **outch** to print a single character and a procedure **outval** to print a value.

```
procedure walk(first)   # print out an expression tree

integer first           # index of root node
integer currentnode
integer stackdepth
common(nodes) struct
                {
                character(1) op
                integer leftp, rightp
                real val
                } tree(100)  # array of structures

struct
                {
                integer nextstate
                integer nodep
                } stackframe(100)

define NODE             tree(currentnode)
define STACK            stackframe(stackdepth)

#  nextstate values
define DOWN             1
define LEFT             2
define RIGHT            3

#  initialize stack with root node
stackdepth = 1
STACK.nextstate = DOWN
STACK.nodep = first
```

```
while( stackdepth > 0 )
      {
      currentnode = STACK.nodep
      select(STACK.nextstate)
            {
            case DOWN:
                  if(NODE.op == " ")  # a leaf
                        {
                        outval( NODE.val )
                        stackdepth -= 1
                        }
                  else  {  # a binary operator node
                        outch( "(" )
                        STACK.nextstate = LEFT
                        stackdepth += 1
                        STACK.nextstate = DOWN
                        STACK.nodep = NODE.leftp
                        }

            case LEFT:
                  outch( NODE.op )
                  STACK.nextstate = RIGHT
                  stackdepth += 1
                  STACK.nextstate = DOWN
                  STACK.nodep = NODE.rightp

            case RIGHT:
                  outch( ")" )
                  stackdepth -= 1
            }
      }
end
```

12. PORTABILITY

One of the major goals of the EFL language is to make it easy to write portable programs. The output of the EFL compiler is intended to be acceptable to any Standard Fortran compiler (unless the **fortran77** option is specified).

12.1. Primitives

Certain EFL operations cannot be implemented in portable Fortran, so a few machine-dependent procedures must be provided in each environment.

12.1.1. Character String Copying

The subroutine **ef1asc** is called to copy one character string to another. If the target string is shorter than the source, the final characters are not copied. If the target string is longer, its end is padded with blanks. The calling sequence is

```
subroutine ef1asc(a, la, b, lb)
integer a(*), la, b(*), lb
```

and it must copy the first **lb** characters from **b** to the first **la** characters of **a**.

12.1.2. Character String Comparisons

The function **eflcmc** is invoked to determine the order of two character strings. The declaration is

```
integer function eflcmc(a, la, b, lb)
integer a(*), la, b(*), lb
```

The function returns a negative value if the string **a** of length **la** precedes the string **b** of length **lb**. It returns zero if the strings are equal, and a positive value otherwise. If the strings are of differing length, the comparison is carried out as if the end of the shorter string were padded with blanks.

13. ACKNOWLEDGMENTS

A. D. Hall originated the EFL language and wrote the first compiler for it; he also gave inestimable aid when I took up the project. B. W. Kernighan and W. S. Brown made a number of useful suggestions about the language and about this report. N. L. Schryer has acted as willing, cheerful, and severe first user and helpful critic of each new version and facility. J. L. Blue, L. C. Kaufman, and D. D. Warner made very useful contributions by making serious use of the compiler, and noting and tolerating its misbehaviors.

14. REFERENCE

1. B. W. Kernighan, "Ratfor — A Preprocessor for a Rational Fortran", Bell Laboratories Computing Science Technical Report #55

APPENDIX A. Relation Between EFL and Ratfor

There are a number of differences between Ratfor and EFL, since EFL is a defined language while Ratfor is the union of the special control structures and the language accepted by the underlying Fortran compiler. Ratfor running over Standard Fortran is almost a subset of EFL. Most of the features described in the Atavisms section are present to ease the conversion of Ratfor programs to EFL.

There are a few incompatibilities: The syntax of the **for** statement is slightly different in the two languages: the three clauses are separated by semicolons in Ratfor, but by commas in EFL. (The initial and iteration statements may be compound statements in EFL because of this change). The input/output syntax is quite different in the two languages, and there is no FORMAT statement in EFL. There are no ASSIGN or assigned GOTO statements in EFL.

The major linguistic additions are character data, factored declaration syntax, block structure, assignment and sequential test operators, generic functions, and data structures. EFL permits more general forms for expressions, and provides a more uniform syntax. (One need not worry about the Fortran/Ratfor restrictions on subscript or DO expression forms, for example.)

APPENDIX B. COMPILER

B.1. Current Version

The current version of the EFL compiler is a two-pass translator written in portable C. It implements all of the features of the language described above except for **long complex** numbers. Versions of this compiler run under the and UNIX† operating systems.

B.2. Diagnostics

The EFL compiler diagnoses all syntax errors. It gives the line and file name (if known) on which the error was detected. Warnings are given for variables that are used but not explicitly declared.

B.3. Quality of Fortran Produced

The Fortran produced by EFL is quite clean and readable. To the extent possible, the variable names that appear in the EFL program are used in the Fortran code. The bodies of loops and test constructs are indented. Statement numbers are consecutive. Few unneeded GOTO and CONTINUE statements are used. It is considered a compiler bug if incorrect Fortran is produced (except for escaped lines). The following is the Fortran procedure produced by the EFL compiler for the matrix multiplication example (Section 11.2):

```
      subroutine matmul(a, b, c, m, n, p)
      integer m, n, p
      double precision a(m, n), b(n, p), c(m, p)
      integer i, j, k
      do 3 i = 1, m
         do 2 j = 1, p
            c(i, j) = 0
            do 1 k = 1, n
               c(i, j) = c(i, j)+a(i, k)*b(k, j)
1              continue
2           continue
3        continue
      end
```

The following is the procedure for the tree walk (Section 11.4):

† UNIX is a trademark of Bell Laboratories.

```
      subroutine walk(first)
      integer first
      common /nodes/ tree
      integer tree(4, 100)
      real tree1(4, 100)
      integer staame(2, 100), stapth, curode
      integer const1(1)
      equivalence (tree(1,1), tree1(1,1))
      data const1(1)/4h     /
c print out an expression tree
c index of root node
c array of structures
c   nextstate values
c   initialize stack with root node
      stapth = 1
      staame(1, stapth) = 1
      staame(2, stapth) = first
    1 if (stapth .le. 0) goto  9
         curode = staame(2, stapth)
         goto  7
    2       if (tree(1, curode) .ne. const1(1)) goto 3
               call outval(tree1(4, curode))
c a leaf
               stapth = stapth-1
               goto  4
    3          call outch(1h()
c a binary operator node
               staame(1, stapth) = 2
               stapth = stapth+1
               staame(1, stapth) = 1
               staame(2, stapth) = tree(2, curode)
    4       goto  8
    5       call outch(tree(1, curode))
            staame(1, stapth) = 3
            stapth = stapth+1
            staame(1, stapth) = 1
            staame(2, stapth) = tree(3, curode)
            goto  8
    6       call outch(1h))
            stapth = stapth-1
            goto  8
    7       if (staame(1, stapth) .eq. 3) goto  6
            if (staame(1, stapth) .eq. 2) goto  5
            if (staame(1, stapth) .eq. 1) goto  2
    8    continue
         goto  1
    9 continue
      end
```

APPENDIX C. CONSTRAINTS ON THE DESIGN OF THE EFL LANGUAGE

Although Fortran can be used to simulate any finite computation, there are realistic limits on the generality of a language that can be translated into Fortran. The design of EFL was constrained by the implementation strategy. Certain of the restrictions are petty (six character external names), but others are sweeping (lack of pointer variables). The following paragraphs describe the major limitations imposed by Fortran.

C.1. External Names

External names (procedure and COMMON block names) must be no longer than six characters in Fortran. Further, an external name is global to the entire program. Therefore, EFL can support block structure within a procedure, but it can have only one level of external name if the EFL procedures are to be compilable separately, as are Fortran procedures.

C.2. Procedure Interface

The Fortran standards, in effect, permit arguments to be passed between Fortran procedures either by reference or by copy-in/copy-out. This indeterminacy of specification shows through into EFL. A program that depends on the method of argument transmission is illegal in either language.

There are no procedure-valued variables in Fortran: a procedure name may only be passed as an argument or be invoked; it cannot be stored. Fortran (and EFL) would be noticeably simpler if a procedure variable mechanism were available.

C.3. Pointers

The most grievous problem with Fortran is its lack of a pointer-like data type. The implementation of the compiler would have been far easier if certain hard cases could have been handled by pointers. Further, the language could have been simplified considerably if pointers were accessible in Fortran. (There are several ways of simulating pointers by using subscripts, but they founder on the problems of external variables and initialization.)

C.4. Recursion

Fortran procedures are not recursive, so it was not practical to permit EFL procedures to be recursive. (Recursive procedures with arguments can be simulated only with great pain.)

C.5. Storage Allocation

The definition of Fortran does not specify the lifetime of variables. It would be possible but cumbersome to implement stack or heap storage disciplines by using COMMON blocks.

Berkeley FP User's Manual, Rev. 4.1

by

Scott Baden

ABSTRACT

This manual describes the Berkeley implementation of Backus' Functional Programming Language, FP. Since this implementation differs from Backus' original description of the language, those familiar with the literature will need to read about the system commands and the local modifications.

June 23, 1987

Table of Contents

1. Background

FP stands for a *Functional Programming* language. Functional programs deal with *functions* instead of *values*. There is no explicit representation of state, there are no assignment statments, and hence, no variables. Owing to the lack of state, FP functions are free from side-effects; so we say the FP is *applicative*.

All functions take one argument and they are evaluated using the single FP operation, *application* (the colon ':' is the apply operator). For example, we read +:<3 4> as "apply the function '+' to its argument <3 4>".

Functional programs express a functional-level combination of their components instead of describing state changes using value-oriented expressions. For example, we write the function returning the *sin* of the *cos* of its input, i.e., *sin* (*cos* (*x*)), as: sin@cos. This is a *functional expression*, consisting of the single *combining form* called *compose* ('@' is the compose operator) and its *functional arguments sin* and *cos*.

All combining forms take functions as arguments and return functions as results; functions may either be applied, e.g., sin@cos:3, or used as a functional argument in another functional expression, e.g., *tan @ sin @ cos*.

As we have seen, FP's combining forms allow us to express control abstractions without the use of variables. The *apply to all* functional form (&) is another case in point. The function '& exp' exponentiates all the elements of its argument:

&exp : <1.0 2.0> ≡ <2.718 7.389> (1.1)

In (1.1) there are no induction variables, nor a loop bounds specification. Moreover, the code is useful for any size argument, so long as the sub-elements of its argument conform to the domain of the *exp* function.

We must change our view of the programming process to adapt to the functional style. Instead of writing down a set of steps that manipulate and assign values, we compose functional expressions using the higher-level functional forms. For example, the function that adds a scalar to all elements of a vector will be written in two steps. First, the function that distributes the scalar amongst each element of the vector:

distl : <3 <4 6>> ≡ <<3 4> <3 6>> (1.2)

Next, the function that adds the pairs of elements that make up a sequence:

&+ : <<3 4> <3 6>> ≡ <7 9> (1.3)

In a value-oriented programming language the computation would be expressed as:

&+ : distl : <3 <4 6>>, (1.4)

which means to apply 'distl' to the input and then to apply '+' to every element of the result. In FP we write (1.4) as:

&+ @ distl : <3 <4 6>>. (1.5)

The functional expression of (1.5) replaces the two step value expression of (1.4).

Often, functional expressions are built from the inside out, as in LISP. In the next example we derive a function that scales then shifts a vector, i.e., for scalars a, b and a vector $\vec{v}$, compute $a + b\vec{v}$. This FP function will have three arguments, namely a, b and $\vec{v}$. Of course, FP does not use formal parameter names, so they will be designated by the function symbols 1, 2, 3. The first code segment scales $\vec{v}$ by b (defintions are delimited with curly braces '{}'):

{scaleVec & @ distl @ [2,3]}* (1.6)

The code segment in (1.5) shifts the vector. The completed function is:

{changeVec &+ @ distl @ [1 , scaleVec]} (1.7)

In the derivation of the program we wrote from right to left, first doing the *distl*'s and then composing with the *apply-to-all* functional form. Using an imperative language, such as Pascal, we would write the program from the outside in, writing the loop before inserting the arithmetic operators.

Although FP encourages a recursive programming style, it provides combining forms to avoid explicit recursion. For example, the right insert combining form (!) can be used to write a function that adds up a list of numbers:

!+ : <1 2 3> ≡ 6 (1.8)

The equivalent, recursive function is much longer:

{addNumbers (null -> %0 ; + @ [1, addNumbers @ tl])} (1.9)

The generality of the combining forms encourages hierarchical program development. Unlike APL, which restricts the use of combining forms to certain builtin functions, FP allows combining forms to take any functional expression as an argument.

2. System Description

2.1. Objects

The set of objects Ω consists of the atoms and sequences $<x_1, x_2, \ldots, x_k>$ (where the $x_i \in \Omega$). (Lisp users should note the similarity to the list structure syntax, just replace the parenthesis by angle brackets and commas by blanks. There are no 'quoted' objects, i.e., 'abc). The atoms uniquely determine the set of valid objects and consist of the numbers (of the type found in FRANZ LISP [Fod80]), quoted ascii strings ("abcd"), and unquoted alphanumeric strings (abc3). There are three predefined atoms, T and F, that correspond to the logical values 'true' and 'false', and the undefined atom ?, *bottom*. *Bottom* denotes the value returned as the result of an undefined operation, e.g., division by zero. The empty sequence, <> is also an atom. The following are examples of valid FP objects:

?	1.47	3888888888888
ab	"*CD*"	<1,<2,3>>
<>	T	*<a,<>>*

There is one restriction on object construction: no object may contain the undefined atom, such an object is itself undefined, e.g., $<1,?> \equiv ?$. This property is the so-called "bottom preserving property" [Ba78].

2.2. Application

This is the single FP operation and is designated by the colon (":"). For a function σ and an object x, $\sigma{:}x$ is an application and its meaning is the object that results from applying σ to x (i.e., evaluating $\sigma(x)$). We say that σ is the *operator* and that x is the *operand*. The following are examples of applications:

+:<7,8>	$\equiv$	15	tl:<1,2,3>	$\equiv$	<2,3>
$1{:}<a,b,c,d>$	$\equiv$	a	$2{:}<a,b,c,d>$	$\equiv$	b

2.3. Functions

All functions (*F*) map objects into objects, moreover, they are *strict*:

$$\sigma{:}? \equiv ?,\ \forall\ \sigma \in \mathbf{F} \qquad (2.1)$$

To formally characterize the primitive functions, we use a modification of McCarthy's conditional expressions [Mc60]:

$$p_1 \rightarrow e_1 ; \cdots ; p_n \rightarrow e_n ; e_{n+1} \qquad (2.2)$$

This statement is interpreted as follows: return function e_1 if the predicate 'p_1' is true , . . . , e_n if 'p_n' is true. If none of the predicates are satisfied then default to e_{n+1}. It is assumed that $x, x_i, y, y_i, z_i \in \Omega$.

2.3.1. Structural

Selector Functions

For a nonzero integer μ,

$\mu : x \equiv$

$x{=}{<}x_1, x_2, \ldots, x_k{>} \wedge 0 < \mu \le k \rightarrow x_\mu;$

$x{=}{<}x_1, x_2, \ldots, x_k{>} \wedge -k{\le}\mu{<}0 \rightarrow x_{k+\mu+1};\ ?$

pick : $<n,x> \equiv$

$x{=}{<}x_1, x_2, \ldots, x_k{>} \wedge 0 < n \le k \rightarrow x_n;$

$x{=}{<}x_1, x_2, \ldots, x_k{>} \wedge -k{\le}n{<}0 \rightarrow x_{k+n+1};\ ?$

The user should note that the function symbols 1,2,3,... are to be distinguished from the atoms 1,2,3,....

last : $x \equiv$

$x{=}{<}{>} \rightarrow {<}{>}\ ;$

$x{=}{<}x_1, x_2, \ldots, x_k{>} \wedge k{\ge}1 \rightarrow x_k;\ ?$

first : $x \equiv$

$x{=}{<}{>} \rightarrow {<}{>}\ ;$

$x{=}{<}x_1, x_2, \ldots, x_k{>} \wedge k{\ge}1 \rightarrow x_1;\ ?$

Tail Functions

tl : $x \equiv$

$x{=}{<}x_1{>} \rightarrow {<}{>}\ ;$

$x{=}{<}x_1, x_2, \ldots, x_k{>} \wedge k{\ge}2 \rightarrow {<}x_2, \ldots, x_k{>}\ ;\ ?$

tlr : $x \equiv$

$x{=}{<}x_1{>} \rightarrow {<}{>}\ ;$

$x{=}{<}x_1, x_2, \ldots, x_k{>} \wedge k{\ge}2 \rightarrow {<}x_1, \ldots, x_{k-1}{>}\ ;\ ?$

Note: There is also a function **front** that is equivalent to **tlr**.

Distribute from left and right

distl : $x \equiv$

$x{=}{<}y,{<}{>}{>} \rightarrow {<}{>};$

$x{=}{<}y,{<}z_1, z_2, \ldots, z_k{>}{>} \rightarrow {<}{<}y,z_1{>},\ldots,{<}y,z_k{>}{>};\ ?$

distr : $x \equiv$

x=<<>,y> → <>;

x=<<y_1, y_2, . . . , y_k>,z> → <<y_1,z>,...,<y_k,z>>; *?*

Identity

id : $x \equiv x$

out : $x \equiv x$

Out is similar to id. Like id it returns its argument as the result, unlike id it prints its result on *stdout* – It is the only function with a side effect. *Out* is intended to be used for debugging only.

Append left and right

apndl : $x \equiv$

x=<y,<>> → <y>;

x=<y,<z_1, z_2, . . . , z_k>> → <y,z_1, z_2, . . . , z_k>; *?*

apndr : $x \equiv$

x=<<>,z> → <z>;

x=<<y_1, y_2, . . . , y_k>,z> → <y_1, y_2, . . . , y_k, z>; *?*

Transpose

trans : $x \equiv$

x=<<>,...,<>> → <>;

x=<x_1, x_2, . . . , x_k> → <y_1, . . . ,y_m>; *?*

where x_i = <x_{i1}, . . . ,x_{im}> ⋌ y_j = <x_{1j}, . . . ,x_{kj}>,
$1 \le i \le k$, $1 \le j \le m$.

reverse : $x \equiv$

x=<> →;

x=<x_1, x_2, . . . , x_k> → <x_k, . . . ,x_1>; *?*

Rotate Left and Right

rotl : $x \equiv$

x=<> → <>; x=<x_1> → <x_1>;

x=<x_1, x_2, . . . , x_k> ⋌ $k \ge 2$ → <x_2, . . . ,x_k,x_1>; *?*

rotr :$x \equiv$

x=<> → <>; x=<x_1> → <x_1>;

x=<x_1, x_2, . . . , x_k> ⋌ $k \ge 2$ → <x_k, x_1, . . . ,x_{k-2}, x_{k-1}>; *?*

concat : x ≡

x=<<$x_{11}, \ldots, x_{1k}$>,<$x_{21}, \ldots, x_{2n}$>,...,<$x_{m1}, \ldots, x_{mp}$>> χ $k, m, n, p > 0$ →
<$x_{11}, \ldots, x_{1k}, x_{21}, \ldots, x_{2n}, \ldots, x_{m1}, \ldots, x_{mp}$>;?

Concatenate removes all occurrences of the null sequence:

concat : <<1,3>,<>,<2,4>,<>,<5>> ≡ <1,3,2,4,5> (2.3)

pair : x ≡

x=<$x_1, x_2, \ldots, x_k$> χ k>0 χ k *is even* → <<x_1,x_2>, . . . ,<x_{k-1},x_k>>;
x=<$x_1, x_2, \ldots, x_k$> χ k>0 χ k *is odd* → <<x_1,x_2>, . . . ,<x_k>>; ?

split : x ≡

x=<x_1> → <<x_1>,<>>;
x=<$x_1, x_2, \ldots, x_k$> χ k>1 → <<$x_1, \ldots, x_{\lceil k/2 \rceil}$>,<$x_{\lceil k/2 \rceil +1}, \ldots, x_k$>>;?

iota :x ≡

x=0 → <>;
$x \in N^+$ → <1,2,...,x>; ?

2.3.2. Predicate (Test) Functions

atom : x ≡ x ∈ *atoms* → T; x≠? → F; ?

eq : x ≡ x =<y,z> χ y=z → T; x=<y,z> χ $y \neq z$ → F; ?

Also less than (<), greater than (>), greater than or equal (>=), less than or equal (<=), not equal (˜=); '=' is a synonym for eq.

null :x ≡ x=<> → T; x≠? → F; ?

length : x ≡ x = <$x_1, x_2, \ldots, x_k$> → k; x=<> → 0; ?

2.3.3. Arithmetic/Logical

+ : x ≡ x=<y,z> χ y,z *are numbers* → $y+z$; ? − : x ≡ x=<y,z> χ y,z *are numbers* → $y-z$; ?
***** : x ≡ x=<y,z> χ y,z *are numbers* → $y \times z$; ? / : x ≡ x=<y,z> χ y,z *are numbers* χ $z \neq 0$ → $y+z$; ?

And, or, not, xor

and : <x,y> ≡ x=T → y; x=F → F; ?

or : <x,y> ≡ x=F → y; x=T → T; ?

xor : $<x,y> \equiv$
x=T χ y=T $\rightarrow$ F; x=F χ y=F $\rightarrow$ F;
x=T χ y=F $\rightarrow$ T; x=F χ y=T $\rightarrow$ T; *?*

not : $x \equiv x$=T $\rightarrow F$; x=F $\rightarrow$ T; *?*

2.3.4. Library Routines

sin : $x \equiv x$ is a number $\rightarrow sin(x)$; *?*

asin : $x \equiv x$ is a number χ $|x| \leq 1 \rightarrow \sin^{-1}(x)$; *?*

cos : $x \equiv x$ is a number $\rightarrow cos(x)$; *?*

acos : $x \equiv x$ is a number χ $|x| \leq 1 \rightarrow \cos^{-1}(x)$; *?*

exp : $x \equiv x$ is a number $\rightarrow e^{x}$; *?*

log : $x \equiv x$ is a positive number $\rightarrow ln(x)$; *?*

mod : $<x,y> \equiv x$ and y are numbers $\rightarrow x - y \times \left\lfloor \frac{x}{y} \right\rfloor$; *?*

2.4. Functional Forms

Functional forms define new *functions* by operating on function and object *parameters* of the form. The resultant expressions can be compared and contrasted to the *value*-oriented expressions of traditional programming languages. The distinction lies in the domain of the operators; functional forms manipulate functions, while traditional operators manipulate values.

One functional form is *composition*. For two functions ϕ and ψ the form ϕ @ ψ denotes their composition $\phi \circ \psi$:

$$(\phi \text{ @ } \psi) : x \equiv \phi:(\psi:x), \ \forall \ \ x \in \Omega \qquad (2.4)$$

The *constant* function takes an object parameter:

$$\%x:y \equiv y=? \rightarrow ?; x, \ \forall \ \ x,y \in \Omega \qquad (2.5)$$

The function %*?* always returns ?.

In the following description of the functional forms, we assume that ξ, ξ_i, σ, σ_i, τ, and τ_i are functions and that x, x_i, y are objects.

Composition

$(\sigma$ @ $\tau):x \equiv \sigma:(\tau:x)$

Construction

$[\sigma_1, \ldots, \sigma_n]:x \equiv <\sigma_1:x,\ldots,\sigma_n:x>$

Note that construction is also bottom-preserving, e.g.,

[+,/] : <3,0> = <3,*?* > = *?* (2.6)

Condition

$(\xi$ -> $\sigma; \tau):x \equiv$
$\quad(\xi:x)=\mathrm{T} \rightarrow \sigma:x;$
$\quad(\xi:x)=\mathrm{F} \rightarrow \tau:x;$ *?*

The reader should be aware of the distinction between *functional expressions*, in the variant of McCarthy's conditional expression, and the *functional form* introduced here. In the former case the result is a *value*, while in the latter case the result is a *function*. Unlike Backus' FP, the conditional form *must* be enclosed in parenthesis, e.g.,

(isNegative -> - @ [%0,id] ; id) (2.7)

Constant

$\%x:y \equiv y=? \rightarrow ?\,; x, \quad \forall\ x \in \Omega$

This function returns its object parameter as its result.

Right Insert

$!\sigma:x \equiv$
$\quad x=<> \rightarrow e_f:x;$
$\quad x=<x_1> \rightarrow x_1;$
$\quad x=<x_1, x_2, \ldots, x_k> \wedge k>2 \rightarrow \sigma:<x_1, !\sigma:<x_2, \ldots, x_k>>;$ *?*

e.g., !+:<4,5,6>=15.

If σ has a right identity element e_f, then $!\sigma:<> = e_f$, e.g.,

!+ : <>=0 and !* : <>=1 (2.8)

Currently, identity functions are defined for + (0), – (0), * (1), / (1), also for **and** (T), **or** (F), **xor** (F). All other unit functions default to bottom (?).

Tree Insert

$|\sigma:x \equiv$
$\quad x=<> \rightarrow e_f:x;$
$\quad x=<x_1> \rightarrow x_1;$
$\quad x=<x_1, x_2, \ldots, x_k> \wedge k>1 \rightarrow$
$\quad \sigma:<|\sigma:<x_1, \ldots, x_{\lceil k/2 \rceil}>, |\sigma:<x_{\lceil k/2 \rceil+1}, \ldots, x_k>>;$*?*

e.g.,

|+:<4,5,6,7> ≡ +:<+:<4,5>,+:<6,7>> ≡ 15 (2.9)

Tree insert uses the same identity functions as right insert.

Apply to All

& $\sigma : x \equiv$

$x=<> \rightarrow <>$;

$x=<x_1, x_2, \ldots, x_k> \rightarrow <\sigma{:}x_1, \ldots, \sigma{:}x_k>$; ?

While

(while ξ σ**)**$:x \equiv$

$\xi{:}x=$**T** $\rightarrow$ **(while** ξ σ**)**$:(\sigma{:}x)$;

$\xi{:}x=$**F** $\rightarrow x$; ?

2.5. User Defined Functions

An FP definition is entered as follows:

{fn-name fn-form}, (2.10)

where *fn-name* is an ascii string consisting of letters, numbers and the underline symbol, and *fn-form* is any valid functional form, including a single primitive or defined function. For example the function

{factorial ! @ iota}* (2.11)

is the non-recursive definition of the factorial function. Since FP systems are applicative it is permissible to substitute the actual definition of a function for any reference to it in a functional form: if $f \equiv 1@2$ then $f : x \equiv 1@2 : x, \;\forall\; x \in \Omega$.

References to undefined functions bottom out:

$$f : x \equiv ?\, \forall\; x \in \Omega, f \notin F \qquad (2.12)$$

3. Getting on and off the System

Startup FP from the shell by entering the command:

/usr/local/fp.

The system will prompt you for input by indenting over six character positions. Exit from FP (back to the shell) with a control/D (^D).

3.1. Comments

A user may end any line (including a command) with a comment; the comment character is '#'. The interpreter will ignore any character after the '#' until it encounters a newline character or end-of-file, whichever comes first.

3.2. Breaks

Breaks interrupt any work in progress causing the system to do a FRANZ reset before returning control back to the user.

3.3. Non-Termination

LISP's namestack may, on occasion, overflow. FP responds by printing "non-terminating" and returning bottom as the result of the application. It does a FRANZ reset before returning control to the user.

4. System Commands

System commands start with a right parenthesis and they are followed by the command-name and possibly one or more arguments. All this information *must be typed on a single line,* and any number of spaces or tabs may be used to separate the components.

4.1. Load

Redirect the standard input to the file named by the command's argument. If the file doesn't exist then FP appends '.fp' to the file-name and retries the open (error if the file doesn't exist). This command allows the user to read in FP function definitions from a file. The user can also read in applications, but such operation is of little utility since none of the input is echoed at the terminal. Normally, FP returns control to the user on an end-of-file. It will also do so whenever it does a FRANZ reset, e.g., whenever the user issues a break, or whenever the system encounters a non-terminating application.

4.2. Save

Output the source text for all user-defined functions to the file named by the argument.

4.3. Csave and Fsave

These commands output the lisp code for all the user-defined functions, including the original source-code, to the file named by the argument. Csave pretty prints the code, Fsave does not. Unless the user wishes to examine the code, he should use 'fsave'; it is about ten times faster than 'csave', and the resulting file will be about three times smaller.

These commands are intended to be used with the liszt compiler and the 'cload' command, as explained below.

4.4. Cload

This command loads or fasls in the file shown by the argument. First, FP appends a '.o' to the file-name, and attempts a load. Failing that, it tries to load the file named by the argument. If the user outputs his function definitions using fsave or csave, and then compiles them using liszt, then he may fasl in the compiled code and speed up the execution of his defined functions by a factor of 5 to 10.

4.5. Pfn

Print the source text(s) (at the terminal) for the user-defined function(s) named by the argument(s) (error if the function doesn't exist).

4.6. Delete

Delete the user-defined function(s) named by the argument (error if the function doesn't exist).

4.7. Fns

List the names of all user-defined functions in alphabetical order. Traced functions are labeled by a trailing '@' (see § 4.7 for sample output).

4.8. Stats

The "stats" command has several options that help the user manage the collection of dynamic statistics for functions[1] and functional forms. Option names follow the keyword "stats", e.g., ")stats reset".

The statistic package records the frequency of usage for each function and functional form; also the size[2] of all the arguments for all functions and functional expressions. These two measures allow the user to derive the average argument size per call. For functional forms the package tallies the frequency of each functional argument. Construction has an additional statistic that tells the number of functional arguments involved in the construction.

Statistics are gathered whenever the mode is on, except for applications that "bottom out" (i.e., return bottom – ?). Statistic collection slows the system down by ×2 to ×4. The following printout illustrates the use of the statistic package (user input is emboldened):

[1] Measurement of user-defined functions is done with the aid of the trace package, discussed in § 4.9.

[2] "Size" is the top-level length of the argument, for most functions. Exceptions are: *apndl, distl* (top-level length of the second element), *apndr, distr* (top-level length of the first element), and *transpose* (top level length of each top level element).

```
                    )stats on

Stats collection turned on.

                    +:<3 4>
7
                    !* @ iota :3
6
                    )stats print

plus:               times       1

times:              times       2

iota:               times       1

insert:             times       1                   size 3

                             Functional Args
                         Name                       Times
                         times                      1


compos:             times       1                   size 1

                             Functional Args
                         Name                       Times
                         insert                     1
                         iota                                   1
```

4.8.1. On

Enable statistics collection.

4.8.2. Off

Disable statistics collection. The user may selectively collect statistics using the on and off commands.

4.8.3. Print

Print the dynamic statistics at the terminal, or, output them to a file. The latter option requires an additional argument, e.g., '')stats print fooBar'' prints the stats to the file ''fooBar''.

4.8.4. Reset

Reset the dynamic statistics counters. To prevent accidental loss of collected statistics, the system will query the user if he tries to reset the counters without first outputting the data (the system will also query the user if he tries to log out without outputting the data).

4.9. Trace

Enable or disable the tracing and the dynamic measurement of the user defined functions named by the argument(s). The first argument tells whether to turn tracing off or on and the others give the name of the functions affected. The tracing and untracing commands are independent of the dynamic statistics commands. This command is cumulative e.g., ')trace on f1', followed by ')trace on f2' is equivalent to ')trace on f1 f2'.

FP tracer output is similar to the FRANZ tracer output: function entries and exits, call level, the functional argument (remember that FP functions have only one argument!), and the result, are printed at the terminal:

```
          )pfn fact

{fact (eq0 -> %1 ; * @ [id, fact @ s1])}
          )fns

eq0       fact  s1

          )trace on fact
          )fns

eq0       fact@     s1

          fact : 2

1 >Enter> fact [2]
|2 >Enter> fact [1]
| 3 >Enter> fact [0]
| 3 <EXIT<  fact  1
|2 <EXIT<  fact  1
1 <EXIT<  fact  2

2
```

4.10. Timer

FP provides a simple timing facility to time top-level applications. The command ")timer on" puts the system in timing mode, ")timer off" turns the mode off (the mode is initially off). While in timing mode, the system reports CPU time, garbage collection time, and elapsed time, in seconds. The timing output follows the printout of the result of the application.

4.11. Script

Open or close a script file. The first argument gives the option, the second the optional script filename. The "open" option causes a new script-file to be opened and any currently open script file to be closed. If the file cannot be opened, FP sends and error message and, if a script file was already opened, it remains open. The command ")script close" closes an open script file. The user may elect to append script output to the script-file with the append mode.

4.12. Help

Print a short summary of all the system commands:

```
    )help
Commands are:

load <file>                      Redirect input from <file>
save <file>                      Save defined fns in <file>
pfn <fn1> ...                    Print source text of <fn1> ...
delete <fn1> ...                 Delete <fn1> ...
fns                              List all functions
stats on/off/reset/print [file]  Collect and print dynamic stats
trace on/off <fn1> ...           Start/Stop exec trace of <fn1> ...
timer on/of                      Turn timer on/off
script open/close/append         Open or close a script-file
lisp                             Exit to the lisp system (return with '^D')
debug on/off                     Turn debugger output on/off
csave <file>                     Output Lisp code for all user-defined fns
cload <file>                     Load Lisp code from a file (may be compiled)
fsave <file>                     Same as csave except without pretty-printing
```

4.13. Special System Functions

There are two system functions that are not generally meant to be used by average users.

4.13.1. Lisp

This exits to the lisp system. Use "^D" to return to FP.

4.13.2. Debug

Turns the 'debug' flag on or off. The command '')debug on'' turns the flag on, '')debug off'' turns the flag off. The main purpose of the command is to print out the parse tree.

5. Programming Examples

We will start off by developing a larger FP program, *mergeSort*. We measure *mergeSort* using the trace package, and then we comment on the measurements. Following *mergeSort* we show an actual session at the terminal.

5.1. MergeSort

The source code for *mergeSort* is:

```
# Use a divide and conquer strategy
{mergeSort | merge}

{merge atEnd @ mergeHelp @ [[], fixLists]}

# Must convert atomic arguments into sequences
# Atomic arguments occur at the leaves of the execution tree
{fixLists &(atom -> [id] ; id)}

# Merge until one or both input lists are empty
{mergeHelp (while and @ &(not@null) @ 2
                   (firstIsSmaller -> takeFirst ;
                                      takeSecond))}

# Find the list with the smaller first element
{firstIsSmaller < @ [1@1@2, 1@2@2]}

# Take the first element of the first list
{takeFirst [apndr@[1,1@1@2], [tl@1@2, 2@2]]}

# Take the first element of the second list
{takeSecond [apndr@[1,1@2@2], [1@2, tl@2@2]]}

# If one list isn't null, then append it to the
# end of the merged list
{atEnd (firstIsNull -> concat@[1,2@2] ;
                       concat@[1,1@2])}

{firstIsNull null@1@2}
```

The merge sort algorithm uses a divide and conquer strategy; it splits the input in half, recursively sorts each half, and then merges the sorted lists. Of course, all these sub-sorts can execute in parallel, and the tree-insert (|) functional form expresses this concurrency. *Merge* removes successively larger elements from the heads of the two lists (either *takeFirst* or *takeSecond*) and appends these elements to the end of the merged sequence. *Merge* terminates when one sequence is empty, and then *atEnd* appends any remaining non-empty sequence to the end of the merged one.

On the next page we give the trace of the function *merge*, which information we can use to determine the structure of *merge*'s execution tree. Since the tree is well-balanced, many of the *merge*'s could be executed in parallel. Using this trace we can also calculate the average length of the arguments passed to *merge*, or a distribution of argument lengths. This information is useful for determining communication costs.

```
          )trace on merge

          mergeSort : <0 3 -2 1 11 8 -22 -33>
| 3 >Enter> merge [<0 3>]
| 3 <EXIT<  merge  <0 3>
| 3 >Enter> merge [<-2 1>]
| 3 <EXIT<  merge  <-2 1>
|2 >Enter> merge [<<0 3> <-2 1>>]
|2 <EXIT<  merge  <-2 0 1 3>
| 3 >Enter> merge [<11 8>]
| 3 <EXIT<  merge  <8 11>
| 3 >Enter> merge [<-22 -33>]
| 3 <EXIT<  merge  <-33 -22>
|2 >Enter> merge [<<8 11> <-33 -22>>]
|2 <EXIT<  merge  <-33 -22 8 11>
1 >Enter> merge [<<-2 0 1 3> <-33 -22 8 11>>]
1 <EXIT<  merge  <-33 -22 -2 0 1 3 8 11>

<-33 -22 -2 0 1 3 8 11>
```

5.2. FP Session

User input is emboldened, terminal output in Roman script.

```
fp

FP, v. 4.1 11/31/82
         )load ex_man
{all_le}
{sort}
{abs_val}
{find}
{ip}
{mm}
{eq0}
{fact}
{sub1}
{alt_fnd}
{alt_fact}
         )fns

abs_val   all_le   alt_fact   alt_fnd   eq0   fact   find
ip        mm       sort       sub1

         abs_val : 3

3

         abs_val : -3

3

         abs_val : 0

0

         abs_val : <-5 0 66>

?

         &abs_val : <-5 0 66>

<5 0 66>

         )pfn abs_val

{abs_val ((> @ [id,%0]) -> id ; (- @ [%0,id]))}

         [id,%0] : -3

<-3 0>

         [%0,id] : -3

<0 -3>
```

```
        - @ [%0,id] : -3
3
        all_le : <1 3 5 7>
T
        all_le : <1 0 5 7>
F
        )pfn all_le
{all_le ! and @ &<= @ distl @ [1,tl]}
        distl @ [1,tl] : <1 2 3 4>
<<1 2> <1 3> <1 4>>
        &<= @ distl @ [1,tl] : <1 2 3 4>
<T T T>
        ! and : <F T T>
F
        ! and : <T T T>
T
        sort : <3 1 2 4>
<1 2 3 4>
        sort : <1>
<1>
        sort : <>
?
        sort : 4
?
        )pfn sort
{sort (null @ tl -> [1] ; (all_le -> apndl @ [1,sort@tl]; sort@rotl))}
        fact : 3
6
        )pfn fact sub1 eq0
```

```
{fact (eq0 -> %1 ; *@[id , fact@sub1])}

{sub1 -@[id,%1]}

{eq0 = @ [id,%0]}

        &fact : <1 2 3 4 5>

<1 2 6 24 120>

        eq0 : 3

F

        eq0 : <>

F

        eq0 : 0

T

        sub1 : 3

2

        %1 : 3

1

        alt_fact : 3

6

        )pfn alt_fact

{alt_fact !* @ iota}

        iota : 3

<1 2 3>

        !* @ iota : 3

6

        !+ : <1 2 3>

6

        find : <3 <3 4 5>>

T

        find : <<> <3 4 <>>>

T
```

```
        find : <3 <4 5>>

F

        )pfn find

{find (null@2 -> %F ; (=@[1,1@2] -> %T ; find@[1,tl@2]))}

        [1,tl@2] : <3 <3 4 5>>

<3 <4 5>>

        [1,1@2] : <3 <3 4 5>>

<3 3>

        alt_fnd : <3 <3 4 5>>

T

        )pfn alt_fnd

{alt_fnd ! or @ &eq @ distl }

        distl : <3 <3 4 5>>

<<3 3> <3 4> <3 5>>

        &eq @ distl : <3 <3 4 5>>

<T F F>

        !or : <T F T>

T

        !or : <F F F>

F

        )delete alt_fnd

        )fns

abs_val   all_le   alt_fact   eq0   fact   find   ip
mm        sort     sub1

        alt_fnd : <3 <3 4 5>>

alt_fnd not defined

?
        {g g}

{g}
        g : 3
```

```
non-terminating

?

[Return to top level]

FP, v. 4.0 10/8/82
        [+,*] : <3 4>

<7 12>

            [+,* : <3 4>

    syntax error:

    [+,* : <3 4>
        ^

        ip : <<3 4 5> <5 6 7>>

74

        )pfn ip

{ip !+ @ &* @ trans}

        trans : <<3 4 5> <5 6 7>>

<<3 5> <4 6> <5 7>>

        &* @ trans : <<3 4 5> <5 6 7>>

<15 24 35>

        mm : <<<1 0> <0 1>> <<3 4> <5 6>>>

<<3 4> <5 6>>

        )pfn mm

{mm &&ip @ &distl @ distr @[1,trans@2]}

        [1,trans@2] : <<<1 0> <0 1>> <<3 4> <5 6>>>

<<<1 0> <0 1>> <<3 4> <5 6>>>

        distr : <<<1 0> <0 1>> <<3 4> <5 6>>>

<<<1 0> <<3 4> <5 6>>> <<0 1> <<3 4> <5 6>>>>

        &distl : <<<1 0> <<3 4> <5 6>>> <<0 1> <<3 4> <5 6>>>>

<<<<1 0> <3 4>> <<1 0> <5 6>>> <<<0 1> <3 4>> <<0 1> <5 6>>>>
```

```
          &ip @ &dist & distr @ [1,trans @ 2] : <<<1 0> <0 1>> <<3 4> <5 6>>>

    syntax error:

    [+,* : <3 4>
         ^
    &ip @ &distl & distr @ [1,trans @ 2] : <<<1 0> <0 1>> <<3 4> <5 6>>>
                 ^

        &ip @ &distl @ distr @ [1,trans@2] : <<<1 0> <0 1>> <<3 4> <5 6>>>

?
```

6. Implementation Notes

FP was written in 3000 lines of FRANZ LISP [Fod 80]. Table 1 breaks down the distribution of the code by functionality.

Functionality	% (bytes)
compiler	34
user interface	32
dynamic stats	16
primitives	14
miscellaneous	3

Table 1

6.1. The Top Level

The top-level function *runFp* starts up the subsystem by calling the routine *fpMain,* that takes three arguments:

(1) A boolean argument that says whether debugging output will be enabled.

(2) A Font identifier. Currently the only one is supported 'asc (ASCII).

(3) A boolean argument that identifies whether the interpreter was invoked from the shell. If so then all exits from FP return the user back to the shell.

The compiler converts the FP functions into LISP equivalents in two stages: first it forms the parse tree, and then it does the code generation.

6.2. The Scanner

The scanner consists of a main routine, *get_tkn,* and a set of action functions. There exists one set of action functions for each character font (currently only ASCII is supported). All the action functions are named *scan$<font>*, where *<font>* is the specified font, and each is keyed on a particular character (or sometimes a particular character-type – e.g., a letter or a number). *get_tkn* returns the token type, and any ancillary information, e.g., for the token "name" the name itself will also be provided. (See Appendix C for the font-token name correspondences). When a character has been read the scanner finds the action function by doing a

(get 'scan$ <font> <char>)

A syntax error message will be generated if no action exists for the particular character read.

6.3. The Parser

The main parsing function, *parse,* accepts a single argument, that identifies the parsing context, or type of construct being handled. Table 2 shows the valid parsing contexts.

id	construct
top_lev	initial call
constr*$$*	construction
compos*$$*	composition
alpha*$$*	apply-to-all
insert*$$*	insert
ti*$$*	tree insert
arrow*$$*	affirmative clause of conditional
semi*$$*	negative clause of conditional
lparen*$$*	parenthetic expr.
while*$$*	while

Table 2, Valid Parsing Contexts

For each type of token there exists a set of parse action functions, of the name *p$<tkn-name>*. Each parse-action function is keyed on a valid context, and it is looked up in the same manner as scan action functions are looked up. If an action function cannot be found, then there is a syntax error in the source code. Parsing proceeds as follows: initially *parse* is called from the top-level, with the context argument set to *"top_lev"*. Certain tokens cause parse to be recursively invoked using that token as a context. The result is the parse tree.

6.4. The Code Generator

The system compiles FP source into LISP source. Normally, this code is interpreted by the FRANZ LISP system. To speed up the implementation, there is an option to compile into machine code using the *liszt* compiler [Joy 79]. This feature improves performance tenfold, for some programs.

The compiler expands all functional forms into their LISP equivalents instead of inserting calls to functions that generate the code at run-time. Otherwise, *liszt* would be ineffective in speeding up execution since all the functional forms would be executed interpretively. Although the amount of code generated by an expanding compiler is 3 or 4 times greater than would be generated by a non-expanding compiler, even in interpreted mode the code runs twice as quickly as unexpanded code. With *liszt* compilation this performance advantage increases to more than tenfold.

A parse tree is either an atom or a hunk of parse trees. An atomic parse-tree identifies either an fp built-in function or a user defined function. The hunk-type parse tree represents functional forms, e.g., compose or construct. The first element identifies the functional form and the other elements are its functional parameters (they may in turn be functional forms). Table 3 shows the parse-tree formats.

Form	Format
user-defined	<atom>
fp builtin	<atom>
apply-to-all	*{alpha$$* Φ*}*
insert	*{insert$$* Φ*}*
tree insert	*{ti$$* Φ*}*
select	*{select$$* μ*}*
constant	*{constant$$* μ*}*
conditional	*{condit$$* Φ_1 Φ_2 Φ_3*}*
while	*{while$$* Φ_1 Φ_2*}*
compose	*{compos$$* Φ_1 Φ_2*}*
construct	*{constr$$* Φ_1 Φ_2 , ... , Φ_n *nil}*

Note: Φ and the Φ_k are parse-trees and μ is an optionally signed integer constant.

Table 3, Parse-Tree Formats

6.5. Function Definition and Application

Once the code has been generated, then the system defines the function via *putd*. The source code is placed onto a property list, *'sources*, to permit later access by the system commands.

For an application, the indicated function is compiled and then defined, only temporarily, as *tmp$$*. The single argument is read and *tmp$$* is applied to it.

6.6. Function Naming Conventions

When the parser detects a named primitive function, it returns the name *<name>$fp*, where *<name>* is the name that was parsed (all primitive function-names end in *$fp*). See Appendix D for the symbolic (e.g., compose, +) function names.

Any name that isn't found in the list of builtin functions is assumed to represent a user-defined function; hence, it isn't possible to redefine FP primitive functions. FP protects itself from accidental or malicious internal destruction by appending the suffix "*_fp*" to all user-defined function names, before they are defined.

6.7. Measurement Impelementation

This work was done by Dorab Patel at UCLA. Most of the measurement code is in the file 'fpMeasures.l'. Many of the remaining changes were effected in 'primFp.l', to add calls on the measurement package at run-time; to 'codeGen.l', to add tracing of user defined functions; to 'utils.l', to add the new system commands; and to 'fpMain.l', to protect the user from forgetting to output statistics when he leaves FP.

6.7.1. Data Structures

All the statistics are in the property list of the global symbol *Measures*. Associated with each each function (primitive or user-defined, or functional form) is an indicator; the statistics gathered for each function are the corresponding values. The names corresponding to primitive functions and functional forms end in '$fp' and the names corresponding to user-defined functions end in '_fp'. Each of the property values is an association list:

```
(get 'Measures 'rotl$fp) ==> ((times . 0) (size . 0))
```

The car of the pair is the name of the statistic (i.e., times, size) and the cdr is the value. There is one exception. Functional forms have a statistic called funargtyp. Instead of being a dotted pair, it is a list of two elements:

```
(get 'Measures 'compose$fp) ==>
((times . 2) (size . 4) (funargtyp ((select$fp . 2) (sub$fp . 2))))
```

The car is the atom 'funargtyp' and the cdr is an alist. Each element of the alist consists of a functional argument-frequency dotted pair.

The statistic packages uses two other global symbols. The symbol DynTraceFlg is non-nil if dynamic statistics are being collected and is nil otherwise. The symbol TracedFns is a list (initially nil) of the names of the user functions being traced.

6.7.2. Interpretation of Data Structures

6.7.2.1. Times

The number of times this function has been called. All functions and functional forms have this statistic.

6.7.2.2. Size

The sum of the sizes of the arguments passed to this function. This could be divided by the times statistic to give the average size of argument this function was passed. With few exceptions, the size of an object is its top-level length (note: version 4.0 defined the size as the total number of *atoms* in the object); the empty sequence, "<>", has a size of 0 and all other atoms have size of one. The exceptions are: *apndl, distl* (top-level length of the second element), *apndr, distr* (top-level length of the first element), and *transpose* (top level length of each top level element).

This statistic is not collected for some primitive functions (mainly binary operators like +,-,*).

6.7.2.3. Funargno

The number of functional arguments supplied to a functional form.

Currently this statistic is gatherered only for the construction functional form.

6.7.2.4. Funargtyp

How many times the named function was used as a functional parameter to the particular functional form.

6.8. Trace Information

The level number of a call shows the number of steps required to execute the function on an ideal machine (i.e., one with unbounded resources). The level number is calculated under an assumption of infinite resources, and the system evaluates the condition of a conditional before evaluating either of its clauses. The number of functions executed at each level can give an idea of the distribution of parallelism in the given FP program.

7. Acknowledgements

Steve Muchnick proposed the initial construction of this system. Bob Ballance added some of is own insights, and John Foderaro provided helpful hints regarding effective use of the FRANZ LISP system [Fod80]. Dorab Patel [Pat81] wrote the dynamic trace and statistics package and made general improvements to the user interface. Portions of this manual were excerpted from the *COMPCON-83 Digest of Papers*[3].

[3] Scott B. Baden and Dorab R. Patel, "Berkeley FP – Experiences With a Functional Programming Language", © 1982, IEEE.

8. References

[Bac78]
John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *CACM*, Turing Award Lecture, 21, 8 (August 1978), 613-641.

[Fod80]
John K. Foderaro, "The FRANZ LISP Manual," University of California, Berkeley, California, 1980.

[Joy79]
W.N. Joy, O. Babaoglu, "UNIX Programmer's Manual," November 7, 1979, Computer Science Division, University of California, Berkeley, California.

[Mc60]
J. McCarthy, "Recursive Functions of Symbolic expressions and their Computation by Machine," Part I, *CACM* 3, 4 (April 1960), 184-195.

[Pat80]
Dorab Ratan Patel, "A System Organization for Applicative Programming," M.S Thesis, University of California, Los Angeles, California, 1980.

[Pat81]
Dorab Patel, "Functional Language Interpreter User Manual," University of California, Los Angeles, California, 1981.

Appendix A: Local Modifications

1. Character Set Changes

Backus [Ba78] used some characters that do not appear on our ASCII terminals, so we have made the following substitutions:

constant	$\bar{x}$	%*x*
insert	/	!
apply-to-all	α	&
composition	$\circ$	@
arrow	$\rightarrow$	->
empty set	ϕ	<>
bottom	$\perp$	?
divide	$\div$	/
multiply	$\times$	*

2. Syntactic Modifications

2.1. While and Conditional

While and conditional functional expressions *must* be enclosed in parenthesis, e.g.,

(**while** *f g*)

(*p* -> *f* ; *g*)

2.2. Function Definitions

Function definitions are enclosed by curly braces; they consist of a name-definition pair, separated by blanks. For example:

{**fact** !* @ iota}

defines the function **fact** (the reader should recognize this as the non-recursive factorial function).

2.3. Sequence Construction

It is not necessary to separate elements of a sequences with a comma; a blank will suffice:

<1,2,3> ≡ <1 2 3>

For nested sequences, the terminating right angle bracket acts as the delimiter:

<<1,2,3>,<4,5,6>> ≡ <<1 2 3><4 5 6>>

3. User Interface

We have provided a rich set of commands that allow the user to catalog, print, and delete functions, to load them from a file and to save them away. The user may generate script files, dynamically trace and measure functional expression execution, generate debugging output, and, temporarily exit to the FRANZ LISP system. A command must begin with a right parenthesis. Consult Appendix C for a complete description of the command syntax.

Debugging in FP is difficult; all undefined results map to a single atom – *bottom* ("?"). To pinpoint the cause of an error the user can use the special debugging output function, out, or the tracer.

4. Additions and Ommissions

Many relational functions have been added: $<$, $>$, $=$, $\neq$, $\leq$, $\geq$; their syntax is: <, >, =, ~=, <=, >=. Also added are the iota function (This is the APL iota function an n-element sequence of natural numbers) and the exclusive OR ($\oplus$) function.

Several new structural functions have been added: **pair** pairs up successive elements of a sequence, **split** splits a sequence into two (roughly) equal halves, **last** returns the last element of the sequence (<> if the sequence is empty), **first** returns the first element of the sequence (<> if it is empty), and **concat** concatenates all subsequences of a sequence, squeezing out null sequences (<>). **Front** is equivalent to **tlr**. **Pick** is a parameterized form of the selector function; the first component of the argument selects a single element from the second component. **Out** is the only side-effect function; it is equivalent to the **id** function but it also prints its argument out at the terminal. This function is intended to be used only for debugging.

One new functional form has been added, tree insert. This functional form breaks up the the argument into two roughly equal pieces applying itself recursively to the two halves. The functional parameter is applied to the result.

The binary-to-unary functions ('**bu**') has been omitted.

Seven mathematical library functions have been added: sin, cos, asin ($\sin^{-1}$), acos ($\cos^{-1}$), log, exp, and mod (the remainder function)

Appendix B: FP Grammar

I. BNF Syntax

```
fpInput →          (fnDef | application | fpCmd[a])*  | '^D'
fnDef →            '{' name funForm '}'
application →      funForm ':' object
name →             letter (letter | digit | '_')*
nameList →         (name)*
object →           atom | fpSequence | '?'
fpSequence →       '<' (ε | object ((',' | ' ') object)*) '>'
atom →             'T' | 'F' | '<>' | '"' (ascii-char)* '"' | (letter | digit)* | number
funForm →          simpFn  | composition | construction | conditional |
                   constantFn | insertion | alpha | while | '(' funForm ')'
simpFn →           fpDefined | fpBuiltin
fpDefined →        name
fpBuiltin →        selectFn | 'tl' | 'id' | 'atom' | 'not' | 'eq' | relFn | 'null' | 'reverse' |
                   'distl' | 'distr' | 'length' | binaryFn | 'trans' | 'apndl' | 'apndr' |
                   'tlr' | 'rotl' | 'rotr' | 'iota' | 'pair' | 'split' | 'concat' | 'last' | 'libFn'
selectFn →         (ε | '+' | '-') unsignedInteger
relFn →            '<=' | '<' | '=' | '~=' | '>' | '>='
binaryFn →         '+' | '-' | '*' | '/' | 'or' | 'and' | 'xor'
libFn →            'sin' | 'cos' | 'asin' | 'acos' | 'log' | 'exp' | 'mod'
composition →      funForm '@' funForm
construction →     '[' formList ']'
formList →         ε | funForm (',' funForm)*
conditional →      '(' funForm '->' funForm ';' funForm ')'
constantFn →       '%' object
insertion →        '!' funForm | '|' funForm
alpha →            '&' funForm
while →            '(' 'while' funForm funForm  ')'
```

II. Precedences

1.	%, !, &	(highest)
2.	@	
3.	[· · ·]	
4.	-> · · · ; · · ·	
5.	while	(least)

[a] Command Syntax is listed in Appendix C.

Appendix C: Command Syntax

All commands begin with a right parenthesis ("")").

```
)fns
)pfn <nameList>
)load <UNIX file name>
)cload <UNIX file name>
)save <UNIX file name>
)csave <UNIX file name>
)fsave <UNIX file name>
)delete <nameList>
)stats on
)stats off
)stats reset
)stats print [UNIX file name]
)trace on  <nameList>
)trace off <nameList>
)timer on
)timer off
)debug on
)debug off
)script open <UNIX file name>
)script close
)script append <UNIX file name>
)help
)lisp
```

Appendix D: Token-Name Correspondences

Token	Name
[	lbrack$$
]	rbrack$$
{	lbrace$$
}	rbrace$$
(	lparen$$
)	rparen$$
@	compos$$
!	insert$$
|	ti$$
&	alpha$$
;	semi$$
:	colon$$
,	comma$$
+	builtin$$
+ μ[a]	select$$
*	builtin$$
/	builtin$$
=	builtin$$
-	builtin$$
->	arrow$$
- μ	select$$
>	builtin$$
>=	builtin$$
<	builtin$$
<=	builtin$$
~=	builtin$$
%o[b]	constant$$

[a] μ is an optionally signed integer constant.

[b] o is any FP object.

Appendix E: Symbolic Primitive Function Names

The scanner assigns names to the alphabetic primitive functions by appending the string "$fp" to the end of the function name. The following table designates the naming assignments to the non-alphabetic primitive function names.

Function	Name
+	plus$fp
-	minus$fp
*	times$fp
/	div$fp
=	eq$fp
>	gt$fp
>=	ge$fp
<	lt$fp
<=	le$fp
~=	ne$fp

RATFOR — A Preprocessor for a Rational Fortran

Brian W. Kernighan

structured programming, control flow, programming

ABSTRACT

Although Fortran is not a pleasant language to use, it does have the advantages of universality and (usually) relative efficiency. The Ratfor language attempts to conceal the main deficiencies of Fortran while retaining its desirable qualities, by providing decent control flow statements:

- statement grouping
- **if-else** and **switch** for decision-making
- **while, for, do,** and **repeat-until** for looping
- **break** and **next** for controlling loop exits

and some ''syntactic sugar'':

- free form input (multiple statements/line, automatic continuation)
- unobtrusive comment convention
- translation of >, >=, etc., into .GT., .GE., etc.
- **return**(expression) statement for functions
- **define** statement for symbolic parameters
- **include** statement for including source files

Ratfor is implemented as a preprocessor which translates this language into Fortran.

Once the control flow and cosmetic deficiencies of Fortran are hidden, the resulting language is remarkably pleasant to use. Ratfor programs are markedly easier to write, and to read, and thus easier to debug, maintain and modify than their Fortran equivalents.

It is readily possible to write Ratfor programs which are portable to other environments. Ratfor is written in itself in this way, so it is also portable; versions of Ratfor are now running on at least two dozen different types of computers at over five hundred locations.

This paper discusses design criteria for a Fortran preprocessor, the Ratfor language and its implementation, and user experience.

1. INTRODUCTION

Most programmers will agree that Fortran is an unpleasant language to program in, yet there are many occasions when they are forced to use it. For example, Fortran is often the only language thoroughly supported on the local computer. Indeed, it is the closest thing to a universal programming language currently available: with care it is possible to write large, truly portable Fortran programs[1]. Finally, Fortran is often the most ''efficient'' language available, particularly for programs requiring much computation.

But Fortran *is* unpleasant. Perhaps the worst deficiency is in the control flow statements — conditional branches and loops — which express the logic of the program. The conditional statements in Fortran are primitive. The Arithmetic IF forces the user into at least two statement numbers and two (implied) GOTO's; it leads to unintelligible code, and is eschewed by good pro-

This paper is a revised and expanded version of oe published in *Software—Practice and Experience*, October 1975. The Ratfor described here is the one in use on UNIX and GCOS at Bell Laboratories, Murray Hill, N. J.

grammers. The Logical IF is better, in that the test part can be stated clearly, but hopelessly restrictive because the statement that follows the IF can only be one Fortran statement (with some *further* restrictions!). And of course there can be no ELSE part to a Fortran IF: there is no way to specify an alternative action if the IF is not satisfied.

The Fortran DO restricts the user to going forward in an arithmetic progression. It is fine for "1 to N in steps of 1 (or 2 or ...)", but there is no direct way to go backwards, or even (in ANSI Fortran[2]) to go from 1 to N–1. And of course the DO is useless if one's problem doesn't map into an arithmetic progression.

The result of these failings is that Fortran programs must be written with numerous labels and branches. The resulting code is particularly difficult to read and understand, and thus hard to debug and modify.

When one is faced with an unpleasant language, a useful technique is to define a new language that overcomes the deficiencies, and to translate it into the unpleasant one with a preprocessor. This is the approach taken with Ratfor. (The preprocessor idea is of course not new, and preprocessors for Fortran are especially popular today. A recent listing [3] of preprocessors shows more than 50, of which at least half a dozen are widely available.)

2. LANGUAGE DESCRIPTION

Design

Ratfor attempts to retain the merits of Fortran (universality, portability, efficiency) while hiding the worst Fortran inadequacies. The language *is* Fortran except for two aspects. First, since control flow is central to any program, regardless of the specific application, the primary task of Ratfor is to conceal this part of Fortran from the user, by providing decent control flow structures. These structures are sufficient and comfortable for structured programming in the narrow sense of programming without GOTO's. Second, since the preprocessor must examine an entire program to translate the control structure, it is possible at the same time to clean up many of the "cosmetic" deficiencies of Fortran, and thus provide a language which is easier and more pleasant to read and write.

Beyond these two aspects — control flow and cosmetics — Ratfor does nothing about the host of other weaknesses of Fortran. Although it would be straightforward to extend it to provide character strings, for example, they are not needed by everyone, and of course the preprocessor would be harder to implement. Throughout, the design principle which has determined what should be in Ratfor and what should not has been *Ratfor doesn't know any Fortran.* Any language feature which would require that Ratfor really understand Fortran has been omitted. We will return to this point in the section on implementation.

Even within the confines of control flow and cosmetics, we have attempted to be selective in what features to provide. The intent has been to provide a small set of the most useful constructs, rather than to throw in everything that has ever been thought useful by someone.

The rest of this section contains an informal description of the Ratfor language. The control flow aspects will be quite familiar to readers used to languages like Algol, PL/I, Pascal, etc., and the cosmetic changes are equally straightforward. We shall concentrate on showing what the language looks like.

Statement Grouping

Fortran provides no way to group statements together, short of making them into a subroutine. The standard construction "if a condition is true, do this group of things," for example,

```
if (x > 100)
        { call error("x>100"); err = 1;
                return }
```

cannot be written directly in Fortran. Instead a programmer is forced to translate this relatively clear thought into murky Fortran, by stating the negative condition and branching around the group of statements:

```
            if (x .le. 100) goto 10
                    call error(5hx>100)
                    err = 1
                    return
10      ...
```

When the program doesn't work, or when it must be modified, this must be translated back into a clearer form before one can be sure what it does.

Ratfor eliminates this error-prone and confusing back-and-forth translation; the first form *is* the way the computation is written in Ratfor. A group of statements can be treated as a unit by enclosing them in the braces { and }. This is true throughout the language: wherever a single Ratfor statement can be used, there can be several enclosed in braces. (Braces seem clearer and less obtrusive than **begin** and **end** or **do** and **end**, and of course **do** and **end** already have Fortran meanings.)

Cosmetics contribute to the readability of code, and thus to its understandability. The character ">" is clearer than ".GT.", so Ratfor translates it appropriately, along with several other similar shorthands. Although many Fortran compilers permit character strings in quotes (like **"x>100"**), quotes are not allowed in ANSI Fortran, so Ratfor converts it into the right number of **H**'s: computers count better than people do.

Ratfor is a free-form language: statements may appear anywhere on a line, and several may appear on one line if they are separated by semicolons. The example above could also be written as

```
if (x > 100) {
        call error("x>100")
        err = 1
        return
}
```

In this case, no semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise.

Of course, if the statement that follows the **if** is a single statement (Ratfor or otherwise), no braces are needed:

```
if (y <= 0.0 & z <= 0.0)
        write(6, 20) y, z
```

No continuation need be indicated because the statement is clearly not finished on the first line. In general Ratfor continues lines when it seems obvious that they are not yet done. (The continuation convention is discussed in detail later.)

Although a free-form language permits wide latitude in formatting styles, it is wise to pick one that is readable, then stick to it. In particular, proper indentation is vital, to make the logical structure of the program obvious to the reader.

The "else" Clause

Ratfor provides an else statement to handle the construction "if a condition is true, do this thing, *otherwise* do that thing."

```
if (a <= b)
        { sw = 0; write(6, 1) a, b }
else
        { sw = 1; write(6, 1) b, a }
```

This writes out the smaller of **a** and **b**, then the larger, and sets **sw** appropriately.

The Fortran equivalent of this code is circuitous indeed:

```
        if (a .gt. b) goto 10
                sw = 0
                write(6, 1) a, b
                goto 20
10      sw = 1
        write(6, 1) b, a
20      ...
```

This is a mechanical translation; shorter forms exist, as they do for many similar situations. But all translations suffer from the same problem: since they are translations, they are less clear and understandable than code that is not a translation. To understand the Fortran version, one must scan the entire program to make sure that no other statement branches to statements 10 or 20 before one knows that indeed this is an **if-else** construction. With the Ratfor version, there is no question about how one gets to the parts of the statement. The **if-else** is a single unit, which can be read, understood, and ignored if not relevant. The program says what it means.

As before, if the statement following an **if** or an **else** is a single statement, no braces are needed:

```
if (a <= b)
        sw = 0
else
        sw = 1
```

The syntax of the **if** statement is

```
if (legal Fortran condition)
        Ratfor statement
else
        Ratfor statement
```

where the **else** part is optional. The *legal Fortran condition* is anything that can legally go into a Fortran Logical IF. Ratfor does not check this clause, since it does not know enough Fortran to know what is permitted. The *Ratfor statement* is any Ratfor or Fortran statement, or any collection of them in braces.

Nested if's

Since the statement that follows an **if** or an **else** can be any Ratfor statement, this leads immediately to the possibility of another **if** or **else**. As a useful example, consider this problem: the variable **f** is to be set to −1 if **x** is less than zero, to +1 if **x** is greater than 100, and to 0 otherwise. Then in Ratfor, we write

```
if (x < 0)
        f = -1
else if (x > 100)
        f = +1
else
        f = 0
```

Here the statement after the first **else** is another **if-else**. Logically it is just a single statement, although it is rather complicated.

This code says what it means. Any version written in straight Fortran will necessarily be indirect because Fortran does not let you say what you mean. And as always, clever shortcuts may turn out to be too clever to understand a year from now.

Following an **else** with an **if** is one way to write a multi-way branch in Ratfor. In general the structure

```
if (...)
        - - -
else if (...)
        - - -
else if (...)
        - - -
...
else
        - - -
```

provides a way to specify the choice of exactly one of several alternatives. (Ratfor also provides a **switch** statement which does the same job in certain special cases; in more general situations, we have to make do with spare parts.) The tests are laid out in sequence, and each one is followed by the code associated with it. Read down the list of decisions until one is found that is satisfied. The code associated with this condition is executed, and then the entire structure is finished. The trailing **else** part handles the "default" case, where none of the other conditions apply. If there is no default action, this final **else** part is omitted:

```
if (x < 0)
        x = 0
else if (x > 100)
        x = 100
```

if-else ambiguity

There is one thing to notice about complicated structures involving nested **if**'s and **else**'s. Consider

```
if (x > 0)
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
```

There are two **if**'s and only one **else**. Which **if** does the **else** go with?

This is a genuine ambiguity in Ratfor, as it is in many other programming languages. The ambiguity is resolved in Ratfor (as elsewhere) by saying that in such cases the **else** goes with the closest previous un-**else**'ed **if**. Thus in this case, the **else** goes with the inner **if**, as we have indicated by the indentation.

It is a wise practice to resolve such cases by explicit braces, just to make your intent clear. In the case above, we would write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
        else
                write(6, 2) y
}
```

which does not change the meaning, but leaves no doubt in the reader's mind. If we want the other association, we *must* write

```
if (x > 0) {
        if (y > 0)
                write(6, 1) x, y
}
else
        write(6, 2) y
```

The "switch" Statement

The **switch** statement provides a clean way to express multi-way branches which branch on the value of some integer-valued expression. The syntax is

```
switch (expression) {

        case expr1 :
                statements
        case expr2, expr3 :
                statements
        ...
        default:
                statements
}
```

Each **case** is followed by a list of comma-separated integer expressions. The *expression* inside **switch** is compared against the case expressions *expr1*, *expr2*, and so on in turn until one matches, at which time the statements following that **case** are executed. If no cases match *expression*, and there is a **default** section, the statements with it are done; if there is no **default**, nothing is done. In all situations, as soon as some block of statements is executed, the entire **switch** is exited immediately. (Readers familiar with C[4] should beware that this behavior is not the same as the C **switch**.)

The "do" Statement

The **do** statement in Ratfor is quite similar to the DO statement in Fortran, except that it uses no statement number. The statement number, after all, serves only to mark the end of the DO, and this can be done just as easily with braces. Thus

```
        do i = 1, n {
                x(i) = 0.0
                y(i) = 0.0
                z(i) = 0.0
        }
```

is the same as

```
        do 10 i = 1, n
                x(i) = 0.0
                y(i) = 0.0
                z(i) = 0.0
   10   continue
```

The syntax is:

```
do legal-Fortran-DO-text
        Ratfor statement
```

The part that follows the keyword **do** has to be something that can legally go into a Fortran DO statement. Thus if a local version of Fortran allows DO limits to be expressions (which is not currently permitted in ANSI Fortran), they can be used in a Ratfor **do**.

The *Ratfor statement* part will often be enclosed in braces, but as with the **if**, a single statement need not have braces around it. This code sets an array to zero:

```
do i = 1, n
        x(i) = 0.0
```

Slightly more complicated,

```
do i = 1, n
        do j = 1, n
                m(i, j) = 0
```

sets the entire array **m** to zero, and

```
do i = 1, n
        do j = 1, n
                if (i < j)
                        m(i, j) = -1
                else if (i == j)
                        m(i, j) = 0
                else
                        m(i, j) = +1
```

sets the upper triangle of **m** to −1, the diagonal to zero, and the lower triangle to +1. (The operator == is "equals", that is, ".EQ.".) In each case, the statement that follows the **do** is logically a *single* statement, even though complicated, and thus needs no braces.

"break" and "next"

Ratfor provides a statement for leaving a loop early, and one for beginning the next iteration. **break** causes an immediate exit from the **do**; in effect it is a branch to the statement *after* the **do**. **next** is a branch to the bottom of the loop, so it causes the next iteration to be done. For example, this code skips over negative values in an array:

```
do i = 1, n {
        if (x(i) < 0.0)
                next
        process positive element
}
```

break and **next** also work in the other Ratfor looping constructions that we will talk about in the next few sections.

break and **next** can be followed by an integer to indicate breaking or iterating that level of enclosing loop; thus

```
	break 2
```

exits from two levels of enclosing loops, and **break 1** is equivalent to **break. next 2** iterates the second enclosing loop. (Realistically, multi-level **break**'s and **next**'s are not likely to be much used because they lead to code that is hard to understand and somewhat risky to change.)

The "while" Statement

One of the problems with the Fortran DO statement is that it generally insists upon being done once, regardless of its limits. If a loop begins

```
	DO I = 2, 1
```

this will typically be done once with **I** set to 2, even though common sense would suggest that perhaps it shouldn't be. Of course a Ratfor **do** can easily be preceded by a test

```
	if (j <= k)
		do i = j, k {
			---
		}
```

but this has to be a conscious act, and is often overlooked by programmers.

A more serious problem with the DO statement is that it encourages that a program be written in terms of an arithmetic progression with small positive steps, even though that may not be the best way to write it. If code has to be contorted to fit the requirements imposed by the Fortran DO, it is that much harder to write and understand.

To overcome these difficulties, Ratfor provides a **while** statement, which is simply a loop: "while some condition is true, repeat this group of statements". It has no preconceptions about why one is looping. For example, this routine to compute sin(x) by the Maclaurin series combines two termination criteria.

```
real function sin(x, e)
	# returns sin(x) to accuracy e, by
	# sin(x) = x - x**3/3! + x**5/5! - ...

	sin = x
	term = x

	i = 3
	while (abs(term)>e & i<100) {
		term = -term * x**2 / float(i*(i-1))
		sin = sin + term
		i = i + 2
	}

	return
	end
```

Notice that if the routine is entered with **term** already smaller than **e**, the loop will be done *zero times*, that is, no attempt will be made to compute **x******3** and thus a potential underflow is avoided. Since the test is made at the top of a **while** loop instead of the bottom, a special case disappears — the code works at one of its boundaries. (The test **i<100** is the other boundary — making sure the routine stops after some maximum number of iterations.)

As an aside, a sharp character "#" in a line marks the beginning of a comment; the rest of the line is comment. Comments and code can co-exist on the same line — one can make marginal remarks, which is not possible with Fortran's "C in column 1" convention. Blank lines are also permitted anywhere (they are not in Fortran); they should be used to emphasize the natural divisions of a program.

The syntax of the **while** statement is

while (*legal Fortran condition*)
	Ratfor statement

As with the **if**, *legal Fortran condition* is something that can go into a Fortran Logical IF, and *Ratfor statement* is a single statement, which may be multiple statements in braces.

The **while** encourages a style of coding not normally practiced by Fortran programmers. For example, suppose **nextch** is a function which returns the next input character both as a function value and in its argument. Then a loop to find the first non-blank character is just

```
	while (nextch(ich) == iblank)
		;
```

A semicolon by itself is a null statement, which is necessary here to mark the end of the **while**; if it were not present, the **while** would control the next statement. When the loop is broken, **ich** contains the first non-blank. Of course the same code can be written in Fortran as

```
 100    if (nextch(ich) .eq. iblank) goto 100
```

but many Fortran programmers (and a few compilers) believe this line is illegal. The language at one's disposal strongly influences how one thinks about a problem.

The "for" Statement

The **for** statement is another Ratfor loop, which attempts to carry the separation of loop-body from reason-for-looping a step further than the **while.** A **for** statement allows explicit initialization and increment steps as part of the statement. For example, a DO loop is just

```
        for (i = 1; i <= n; i = i + 1) ...
```

This is equivalent to

```
        i = 1
        while (i <= n) {
                ...
                i = i + 1
        }
```

The initialization and increment of **i** have been moved into the **for** statement, making it easier to see at a glance what controls the loop.

The **for** and **while** versions have the advantage that they will be done zero times if **n** is less than 1; this is not true of the **do.**

The loop of the sine routine in the previous section can be re-written with a **for** as

```
      for (i=3; abs(term) > e & i < 100;
        i=i+2) {
              term = -term * x**2 / float(i*(i-1))
              sin = sin + term
      }
```

The syntax of the **for** statement is

for (*init* ; *condition* ; *increment*)
Ratfor statement

init is any single Fortran statement, which gets done once before the loop begins. *increment* is any single Fortran statement, which gets done at the end of each pass through the loop, before the test. *condition* is again anything that is legal in a logical IF. Any of *init, condition,* and *increment* may be omitted, although the semicolons *must* always be present. A non-existent *condition* is treated as always true, so **for(;;)** is an indefinite repeat. (But see the **repeat-until** in the next section.)

The **for** statement is particularly useful for backward loops, chaining along lists, loops that might be done zero times, and similar things which are hard to express with a DO statement, and obscure to write out with IF's and GOTO's. For example, here is a backwards DO loop to find the last non-blank character on a card:

```
        for (i = 80; i > 0; i = i - 1)
                if (card(i) != blank)
                        break
```

("!=" is the same as ".NE."). The code scans the columns from 80 through to 1. If a non-blank is found, the loop is immediately broken. (**break** and **next** work in **for**'s and **while**'s just as in **do**'s). If **i** reaches zero, the card is all blank.

This code is rather nasty to write with a regular Fortran DO, since the loop must go forward, and we must explicitly set up proper conditions when we fall out of the loop. (Forgetting this is a common error.) Thus:

```
        DO 10 J = 1, 80
              I = 81 - J
              IF (CARD(I) .NE. BLANK) GO TO 11
 10     CONTINUE
        I = 0
 11     ...
```

The version that uses the **for** handles the termination condition properly for free; **i** *is* zero when we fall out of the **for** loop.

The increment in a **for** need not be an arithmetic progression; the following program walks along a list (stored in an integer array **ptr**) until a zero pointer is found, adding up elements from a parallel array of values:

```
        sum = 0.0
        for (i = first; i > 0; i = ptr(i))
                sum = sum + value(i)
```

Notice that the code works correctly if the list is empty. Again, placing the test at the top of a loop instead of the bottom eliminates a potential boundary error.

The "repeat-until" statement

In spite of the dire warnings, there are times when one really needs a loop that tests at the bottom after one pass through. This service is provided by the **repeat-until**:

```
repeat
        Ratfor statement
until (legal Fortran condition)
```

The *Ratfor statement* part is done once, then the condition is evaluated. If it is true, the loop is exited; if it is false, another pass is made.

The **until** part is optional, so a bare **repeat** is the cleanest way to specify an infinite loop. Of course such a loop must ultimately be broken by some transfer of control such as **stop**, **return**, or **break**, or an implicit stop such as running out of input with a READ statement.

As a matter of observed fact[8], the **repeat-until** statement is *much* less used than the other looping constructions; in particular, it is typically outnumbered ten to one by **for** and **while**. Be cautious about using it, for loops that test only at the bottom often don't handle null cases well.

More on break and next

break exits immediately from **do**, **while**, **for**, and **repeat-until**. **next** goes to the test part of **do**, **while** and **repeat-until**, and to the increment step of a **for**.

"return" Statement

The standard Fortran mechanism for returning a value from a function uses the name of the function as a variable which can be assigned to; the last value stored in it is the function value upon return. For example, here is a routine **equal** which returns 1 if two arrays are identical, and zero if they differ. The array ends are marked by the special value −1.

```
# equal _ compare str1 to str2;
#       return 1 if equal, 0 if not
        integer function equal(str1, str2)
        integer str1(100), str2(100)
        integer i

        for (i = 1; str1(i) == str2(i); i = i + 1)
                if (str1(i) == -1) {
                        equal = 1
                        return
                }
        equal = 0
        return
        end
```

In many languages (e.g., PL/I) one instead says

```
return (expression)
```

to return a value from a function. Since this is often clearer, Ratfor provides such a **return** statement — in a function **F**, **return**(expression) is equivalent to

```
{ F = expression; return }
```

For example, here is **equal** again:

```
# equal _ compare str1 to str2;
#       return 1 if equal, 0 if not
        integer function equal(str1, str2)
        integer str1(100), str2(100)
        integer i

        for (i = 1; str1(i) == str2(i); i = i + 1)
                if (str1(i) == -1)
                        return(1)
        return(0)
        end
```

If there is no parenthesized expression after **return**, a normal RETURN is made. (Another version of **equal** is presented shortly.)

Cosmetics

As we said above, the visual appearance of a language has a substantial effect on how easy it is to read and understand programs. Accordingly, Ratfor provides a number of cosmetic facilities which may be used to make programs more readable.

Free-form Input

Statements can be placed anywhere on a line; long statements are continued automatically, as are long conditions in **if**, **while**, **for**, and **until**. Blank lines are ignored. Multiple statements may appear on one line, if they are separated by semicolons. No semicolon is needed at the end of a line, if Ratfor can make some reasonable guess about whether the statement ends there. Lines ending with any of the characters

```
= + - * , | & ( _
```

are assumed to be continued on the next line. Underscores are discarded wherever they occur; all others remain as part of the statement.

Any statement that begins with an all-numeric field is assumed to be a Fortran label, and placed in columns 1-5 upon output. Thus

```
write(6, 100); 100 format("hello")
```

is converted into

```
        write(6, 100)
100     format(5hhello)
```

Translation Services

Text enclosed in matching single or double quotes is converted to **nH...** but is otherwise unaltered (except for formatting — it may get split across card boundaries during the reformatting process). Within quoted strings, the backslash '\' serves as an escape character: the next character is taken literally. This provides a way to get quotes (and of course the backslash itself) into quoted strings:

```
"\\\'"
```

is a string containing a backslash and an apostrophe. (This is *not* the standard convention of doubled quotes, but it is easier to use and more general.)

Any line that begins with the character '%' is left absolutely unaltered except for stripping off the '%' and moving the line one position to the left. This is useful for inserting control cards, and other things that should not be transmogrified (like an existing Fortran program). Use '%' only for ordinary statements, not for the condition parts of **if**, **while**, etc., or the output may come out in an unexpected place.

The following character translations are made, except within single or double quotes or on a line beginning with a '%'.

==	.eq.	!=	.ne.
>	.gt.	>=	.ge.
<	.lt.	<=	.le.
&	.and.	|	.or.
!	.not.	¬	.not.

In addition, the following translations are provided for input devices with restricted character sets.

[	{	]	}
$(	{	$)	}

"define" Statement

Any string of alphanumeric characters can be defined as a name; thereafter, whenever that name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest of the definition line. (Comments and trailing white spaces are stripped off). A defined name can be arbitrarily long, and must begin with a letter.

define is typically used to create symbolic parameters:

```
define ROWS     100
define COLS 50

dimension a(ROWS), b(ROWS, COLS)
    if (i > ROWS | j > COLS) ...
```

Alternately, definitions may be written as

```
define(ROWS, 100)
```

In this case, the defining text is everything after the comma up to the balancing right parenthesis; this allows multi-line definitions.

It is generally a wise practice to use symbolic parameters for most constants, to help make clear the function of what would otherwise be mysterious numbers. As an example, here is the routine **equal** again, this time with symbolic constants.

```
define    YES    1
define    NO     0
define    EOS    -1
define    ARB    100

# equal _ compare str1 to str2;
#     return YES if equal, NO if not
      integer function equal(str1, str2)
      integer str1(ARB), str2(ARB)
      integer i

      for (i = 1; str1(i) == str2(i);
      i = i + 1)
            if (str1(i) == EOS)
                  return(YES)
      return(NO)
      end
```

"include" Statement

The statement

```
include file
```

inserts the file found on input stream *file* into the Ratfor input in place of the **include** statement. The standard usage is to place COMMON blocks on a file, and **include** that file whenever a copy is needed:

```
subroutine x
        include commonblocks
        ...
        end

suroutine y
        include commonblocks
        ...
        end
```

This ensures that all copies of the COMMON blocks are identical

Pitfalls, Botches, Blemishes and other Failings

Ratfor catches certain syntax errors, such as missing braces, **else** clauses without an **if**, and most errors involving missing parentheses in statements. Beyond that, since Ratfor knows no Fortran, any errors you make will be reported by the Fortran compiler, so you will from time to time have to relate a Fortran diagnostic back to the Ratfor source.

Keywords are reserved — using **if**, **else**, etc., as variable names will typically wreak havoc. Don't leave spaces in keywords. Don't use the Arithmetic IF.

The Fortran **nH** convention is not recognized anywhere by Ratfor; use quotes instead.

3. IMPLEMENTATION

Ratfor was originally written in C[4] on the UNIX operating system[5]. The language is specified by a context free grammar and the compiler constructed using the YACC compiler-compiler[6].

The Ratfor grammar is simple and straightforward, being essentially

```
prog   : stat
       | prog  stat
stat   : if (...) stat
       | if (...) stat else stat
       | while (...) stat
       | for (...; ...; ...) stat
       | do ... stat
       | repeat stat
       | repeat stat until (...)
       | switch (...) { case ...: prog ...
                   default: prog }
       | return
       | break
       | next
       | digits  stat
       | { prog }
       | anything unrecognizable
```

The observation that Ratfor knows no Fortran follows directly from the rule that says a statement is "anything unrecognizable". In fact most of Fortran falls into this category, since any statement that does not begin with one of the keywords is by definition "unrecognizable."

Code generation is also simple. If the first thing on a source line is not a keyword (like **if**, **else**, etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. (Leading digits are treated as a label.) Keywords cause only slightly more complicated actions. For example, when **if** is recognized, two consecutive labels L and L+1 are generated and the value of L is stacked. The condition is then isolated, and the code

```
if (.not. (condition)) goto L
```

is output. The *statement* part of the **if** is then translated. When the end of the statement is encountered (which may be some distance away and include nested **if**'s, of course), the code

```
L      continue
```

is generated, unless there is an **else** clause, in which case the code is

```
        goto L+1
L       continue
```

In this latter case, the code

```
L+1     continue
```

is produced after the *statement* part of the else. Code generation for the various loops is equally simple.

One might argue that more care should be taken in code generation. For example, if there is no trailing else,

```
        if (i > 0) x = a
```

should be left alone, not converted into

```
        if (.not. (i .gt. 0)) goto 100
        x = a
100     continue
```

But what are optimizing compilers for, if not to improve code? It is a rare program indeed where this kind of "inefficiency" will make even a measurable difference. In the few cases where it is important, the offending lines can be protected by '%'.

The use of a compiler-compiler is definitely the preferred method of software development. The language is well-defined, with few syntactic irregularities. Implementation is quite simple; the original construction took under a week. The language is sufficiently simple, however, that an *ad hoc* recognizer can be readily constructed to do the same job if no compiler-compiler is available.

The C version of Ratfor is used on UNIX and on the Honeywell GCOS systems. C compilers are not as widely available as Fortran, however, so there is also a Ratfor written in itself and originally bootstrapped with the C version. The Ratfor version was written so as to translate into the portable subset of Fortran described in [1], so it is portable, having been run essentially without change on at least twelve distinct machines. (The main restrictions of the portable subset are: only one character per machine word; subscripts in the form $c*v \pm c$; avoiding expressions in places like DO loops; consistency in subroutine argument usage, and in COMMON declarations. Ratfor itself will not gratuitously generate non-standard Fortran.)

The Ratfor version is about 1500 lines of Ratfor (compared to about 1000 lines of C); this compiles into 2500 lines of Fortran. This expansion ratio is somewhat higher than average, since the compiled code contains unnecessary occurrences of COMMON declarations. The execution time of the Ratfor version is dominated by two routines that read and write cards. Clearly these routines could be replaced by machine coded local versions; unless this is done, the efficiency of other parts of the translation process is largely irrelevant.

4. EXPERIENCE

Good Things

"It's so much better than Fortran" is the most common response of users when asked how well Ratfor meets their needs. Although cynics might consider this to be vacuous, it does seem to be true that decent control flow and cosmetics converts Fortran from a bad language into quite a reasonable one, assuming that Fortran data structures are adequate for the task at hand.

Although there are no quantitative results, users feel that coding in Ratfor is at least twice as fast as in Fortran. More important, debugging and subsequent revision are much faster than in Fortran. Partly this is simply because the code can be *read.* The looping statements which test at the top instead of the bottom seem to eliminate or at least reduce the occurrence of a wide class of boundary errors. And of course it is easy to do structured programming in Ratfor; this self-discipline also contributes markedly to reliability.

One interesting and encouraging fact is that programs written in Ratfor tend to be as readable as programs written in more modern languages like Pascal. Once one is freed from the shackles of Fortran's clerical detail and rigid input format, it is easy to write code that is readable, even esthetically pleasing. For example, here is a Ratfor implementation of the linear table search discussed by Knuth [7]:

```
A(m+1) = x
for (i = 1; A(i) != x; i = i + 1)
        ;
if (i > m) {
        m = i
        B(i) = 1
}
else
        B(i) = B(i) + 1
```

A large corpus (5400 lines) of Ratfor, including a subset of the Ratfor preprocessor itself, can be found in [8].

Bad Things

The biggest single problem is that many Fortran syntax errors are not detected by Ratfor but by the local Fortran compiler. The compiler then prints a message in terms of the generated Fortran, and in a few cases this may be difficult to relate back to the offending Ratfor line, especially if the implementation conceals the generated Fortran. This problem could be dealt with by tagging each generated line with some indication of the source line that created it, but this is inherently implementation-dependent, so no action has yet been taken. Error message interpretation is actually not so arduous as might be thought. Since Ratfor generates no variables, only a simple pattern of IF's and GOTO's, data-related errors like missing DIMENSION statements are easy to find in the Fortran. Furthermore, there has been a steady improvement in Ratfor's ability to catch trivial syntactic errors like unbalanced parentheses and quotes.

There are a number of implementation weaknesses that are a nuisance, especially to new users. For example, keywords are reserved. This rarely makes any difference, except for those hardy souls who want to use an Arithmetic IF. A few standard Fortran constructions are not accepted by Ratfor, and this is perceived as a problem by users with a large corpus of existing Fortran programs. Protecting every line with a '%' is not really a complete solution, although it serves as a stop-gap. The best long-term solution is provided by the program Struct [9], which converts arbitrary Fortran programs into Ratfor.

Users who export programs often complain that the generated Fortran is "unreadable" because it is not tastefully formatted and contains extraneous CONTINUE statements. To some extent this can be ameliorated (Ratfor now has an option to copy Ratfor comments into the generated Fortran), but it has always seemed that effort is better spent on the input language than on the output esthetics.

One final problem is partly attributable to success — since Ratfor is relatively easy to modify, there are now several dialects of Ratfor. Fortunately, so far most of the differences are in character set, or in invisible aspects like code generation.

5. CONCLUSIONS

Ratfor demonstrates that with modest effort it is possible to convert Fortran from a bad language into quite a good one. A preprocessor is clearly a useful way to extend or ameliorate the facilities of a base language.

When designing a language, it is important to concentrate on the essential requirement of providing the user with the best language possible for a given effort. One must avoid throwing in "features" — things which the user may trivially construct within the existing framework.

One must also avoid getting sidetracked on irrelevancies. For instance it seems pointless for Ratfor to prepare a neatly formatted listing of either its input or its output. The user is presumably capable of the self-discipline required to prepare neat input that reflects his thoughts. It is much more important that the language provide free-form input so he *can* format it neatly. No one should read the output anyway except in the most dire circumstances.

Acknowledgements

C. A. R. Hoare once said that "One thing [the language designer] should not do is to include untried ideas of his own." Ratfor follows this precept very closely — everything in it has been stolen from someone else. Most of the control flow structures are taken directly from the language C[4] developed by Dennis Ritchie; the comment and continuation conventions are adapted from Altran[10].

I am grateful to Stuart Feldman, whose patient simulation of an innocent user during the early days of Ratfor led to several design improvements and the eradication of bugs. He also translated the C parse-tables and YACC parser into Fortran for the first Ratfor version of Ratfor.

References

[1] B. G. Ryder, "The PFORT Verifier," *Software—Practice & Experience*, October 1974.

[2] American National Standard Fortran. American National Standards Institute, New York, 1966.

[3] *For-word: Fortran Development Newsletter*, August 1975.

[4] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., 1978.

[5] D. M. Ritchie and K. L. Thompson, "The UNIX Time-sharing System." *CACM*, July 1974.

[6] S. C. Johnson, "YACC — Yet Another Compiler-Compiler." Bell Laboratories Computing Science Technical Report #32, 1978.

[7] D. E. Knuth, "Structured Programming with goto Statements." *Computing Surveys*, December 1974.

[8] B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

[9] B. S. Baker, "Struct — A Program which Structures Fortran", Bell Laboratories internal memorandum, December 1975.

[10] A. D. Hall, "The Altran System for Rational Function Manipulation — A Survey." *CACM*, August 1971.

Appendix: Usage on UNIX and GCOS.

Beware — local customs vary. Check with a native before going into the jungle.

UNIX

The program **ratfor** is the basic translator; it takes either a list of file names or the standard input and writes Fortran on the standard output. Options include **–6x**, which uses x as a continuation character in column 6 (UNIX uses & in column 1), and –C, which causes Ratfor comments to be copied into the generated Fortran.

The program **rc** provides an interface to the **ratfor** command which is much the same as cc. Thus

```
rc [options] files
```

compiles the files specified by **files**. Files with names ending in **.r** are Ratfor source; other files are assumed to be for the loader. The flags –C and **–6x** described above are recognized, as are

–c	compile only; don't load
–f	save intermediate Fortran .f files
–r	Ratfor only; implies –c and –f
–2	use big Fortran compiler (for large programs)
–U	flag undeclared variables (not universally available)

Other flags are passed on to the loader.

GCOS

The program **./ratfor** is the bare translator, and is identical to the UNIX version, except that the continuation convention is & in column 6. Thus

```
./ratfor  files  >output
```

translates the Ratfor source on **files** and collects the generated Fortran on file 'output' for subsequent processing.

./rc provides much the same services as **rc** (within the limitations of GCOS), regrettably with a somewhat different syntax. Options recognized by **./rc** include

name	Ratfor source or library, depending on type
h=/name	make TSS H* file (runnable version); run as /name
r=/name	update and use random library
a=	compile as ascii (default is bcd)
C=	copy comments into Fortran
f=name	Fortran source file
g=name	gmap source file

Other options are as specified for the ./cc command described in [4].

TSO, TSS, and other systems

Ratfor exists on various other systems; check with the author for specifics.

Integrated Solutions

AN NBI COMPANY

DOCUMENTATION COMMENTS

Please take a minute to comment on the accuracy and completeness of this manual. Your assistance will help us to better identify and respond to specific documentation issues. If necessary, you may attach an additional page with comments. Thank you in advance for your cooperation.

Manual Title:	*UNIX Prog. Supplement (PS2)*	Part Number:	*490147 Rev. A*

Name: ______________________ Title: ______________________

Company: ______________________ Phone: (   ) ______________________

Address: ____

City: ______________________ State: ______ Zip Code: ____________

1. Please rate this manual for the following:

	Poor	Fair	Good	Excellent
Clarity	☐	☐	☐	☐
Completeness	☐	☐	☐	☐
Organization	☐	☐	☐	☐
Technical Content/Accuracy	☐	☐	☐	☐
Readability	☐	☐	☐	☐

Please comment: ____

____

2. Does this manual contain enough examples and figures?
Yes ☐ No ☐

Please comment: ____

____

3. Is any information missing from this manual?
Yes ☐ No ☐

Please comment: ____

____

4. Is this manual adequate for your purposes?
Yes ☐ No ☐

Please comment on how this manual can be improved: ____

____

____

____

____

____

____

Fold Down First F

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

First-Class Mail Permit No. 7628 San Jose, California 95131

Postage will be paid by addressee

Integrated Solutions

An NBI Company

ATTN: Technical Publications Manager
1140 Ringwood Court
San Jose, CA 95131

ɔld Up Second

Staple Here