```
 ____         __ _                            _____           _     
/ ___|  ___  / _| |___      ____ _ _ __ ___  |_   _|__   ___ | |___ 
\___ \ / _ \| |_| __\ \ /\ / / _` | '__/ _ \   | |/ _ \ / _ \| / __|
 ___) | (_) |  _| |_ \ V  V / (_| | | |  __/   | | (_) | (_) | \__ \
|____/ \___/|_|  \__| \_/\_/ \__,_|_|  \___|   |_|\___/ \___/|_|___/

 __  __                         _     
|  \/  | __ _ _ __  _   _  __ _| |___ 
| |\/| |/ _` | '_ \| | | |/ _` | / __|
| |  | | (_| | | | | |_| | (_| | \__ \
|_|  |_|\__,_|_| |_|\__,_|\__,_|_|___/
```

```
 ____                _ __  __                            _ 
|  _ \ ___  __ _  __| |  \/  | ___  ___    __ _ _ __   __| |
| |_) / _ \/ _` |/ _` | |\/| |/ _ \/ __|  / _` | '_ \ / _` |
|  _ <  __/ (_| | (_| | |  | |  __/\__ \ | (_| | | | | (_| |
|_| \_\___|\__,_|\__,_|_|  |_|\___||___/  \__,_|_| |_|\__,_|

 ____      _                       _   _       _            
|  _ \ ___| | ___  __ _ ___  ___  | \ | | ___ | |_ ___  ___ 
| |_) / _ \ |/ _ \/ _` / __|/ _ \ |  \| |/ _ \| __/ _ \/ __|
|  _ <  __/ |  __/ (_| \__ \  __/ | |\  | (_) | ||  __/\__ \
|_| \_\___|_|\___|\__,_|___/\___| |_| \_|\___/ \__\___||___/
```

Editor’s note:

I had a little trouble building the tools. Here’s how I got out of it:

The release notes tell you to create st_bin, set your default there, and copy all of the files in [.distn] on the distribution kit to st_bin. They don’t tell you that the build procedure expects the files from [.src] and [.vms] on the distribution to be copied to [-.src] and [-.vms].

You can put your st_tmp (~tmp) directory on any device you like, except when doing a toolgen. Toolgen renames files between st_bin and st_tmp, so they have to be on the same physical device. After doing a toolgen, you can reassign st_tmp to wherever you want it to be.

The toolgen procedure does an @fbuild, which builds a few tools that the tools use to build themselves. It assumes that they are all defined as VMS foreign commands. I added the command definitions to fbuild.com.

Some system-wide logical names are defined in stlognam.com, which you should edit for your system. The sharable image’s name RLIBSHARE was assigned /exec but not /system. I added the /system to stlognam.com.

Spring 1986 DECUS Distribution

The following steps must be performed to build the Spring 1986 release of the Software Tools package for VMS.

1. Edit the file stlognam.com in this distribution directory to reflect the disk and directories used by the tools. All of the tools logical names start with the string "st_", in an attempt to avoid conflicts with all other software. The definition for st_node should be replaced with your node name, and st_timezone should be replaced with the appropriate three character mnemonic. Do not worry, the software which uses the logical name worries about whether it is daylight-savings time or not, so you won't have to worry about changing the logical name each April and October.

2. Invoke the modified stlognam.com to set up the environment

3. Create the six known directories (~bin, ~usr, ~tmp, ~lpr, ~msg and ~man) with the appropriate protection, and set default to st_bin. Consult release.doc for information on the required protection modes for the directories.

4. Copy the Distn directory files into the current directory, after deleting all files currently in the directory. Make sure that the account under which you are running has the following quotas:

PRCLM	10
BYTLM	30000
FILLM	75
TQELM	40
PGFLQUOTA	16384

5. @toolgen
   This command file assembles all macro primitives, compiles all fortran primitives, builds two tools to bootstrap ourselves, and then proceeds to build the 115 utilities in the package. This takes a few hours, so take a break. If you answer yes to any of the questions concerning file deletion, toolgen will delete unneeded files as the processing progresses. If you delete the object files, a savings of ~2000 disk blocks ensues. If you delete the source files as you progress, a savings of ~3700 blocks accrues. If a shared global image is NOT selected, the entire system occupies ~22000 blocks if no files are deleted, or ~16000 blocks if both sources and objects are deleted during the build. On the other hand, by building with the shared global image, the corresponding numbers are ~12000 and ~6000, respectively.

6. Now modify the system startup files to setup the new logical names and installed images for the next boot.

7. Install the known images using st_bin:tools.ins

8. The required quotas have not changed with this release, so no mucking with the authorization file will be necessary, unless this is your first tools release. If this is the case, consult the file release.doc in the distn directory.

9. To build the appropriate mail system utilities, you need to consult the file msgreadme.1st in the msgsys directory; follow the directions there.

10. You should now be operational.

Software Tools Users Group 140 Center Street El Segundo, CA 90245

(213) 322-2574

VAX/VMS Software Tools VOS

Spring 1986 DECUS Distribution

David Martin
Hughes Aircraft
P.O. Box 92426 R1/B206
Los Angeles, CA 90009

This document describes the VMS implementation of the Software Tools Virtual Operating System. For those new to this game, the basic principles behind the software are described in the article "A Virtual Operating System" which appeared in the September 1980 issue of the Communications of the ACM. The contents of this release supercede all previous releases.

See the file ``changes.s86'' for a list of changes since the last (s84) release.

NOTICE

This software is provided on an as-is basis. No guarantee of performance or support is stated or implied. Any errors or omissions in the code or documentation are regrettable, but not unusual considering the man-power and mode of distribution. Written notification of bugs WITH fixes are appreciated and will be incorporated into the next release, if possible.

Currently available tools

Acat	- concatenate nested archive entries on standard output
Addr	- generate the msg address database
Admin	- administer TCS file.
Alist	- generate paginated listing of source archive
Ar	- archive file maintainer
Args	- use standard input as arguments for command
Asam	- generate index for archive file
Asplit	- salvage garbaged archive files
Axref	- cross reference symbols in archive files
Banner	- generate large banner lines
BarGraph	- draw a 0-100% bargraph of integer data
Box	- draw boxes around block structure of RatFor or C programs
Cat	- concatenate and print text files
Ccnt	- character count
Cd	- change (current) directory
Ch	- make changes in text files
Chmod	- change mode (protection codes) of file
Chown	- change the ownership of file(s).
Cmp	- compare two files
Comm	- print lines common to two files
Cron	- clock deamon
Cp	- copy files
Cpress	- compress input files
Crt	- copy files to terminal a screen at a time
Crypt	- crypt and decrypt standard input
D	- list contents of directory
Date	- print the date
Dc	- desk calculator
Delta	- make an TCS delta
Detab	- convert tabs to spaces
Diff	- isolate differences between files
E	- extended version of "ed" with command editing & history
Echo	- echo command line arguments
Ed	- line-oriented text editor
Entab	- convert spaces to tabs and spaces
Esh	- extended shell, with intraline editing and history
Exist	- check for the existence of a file
Expand	- uncompress input files
Fb	- search blocks of lines for text patterns
Fc	- fortran compiler
Fd	- fast directory list in sort order
Field	- manipulate fields of data
Find	- search a file for text patterns
Form	- produce form letter by prompting user for information
Format	- format (roff) text
Get	- get generation from TCS file
Grep	- search file[s] for a pattern

Hsh	- shell with history and editing functions
Incl	- expand included files
Intro	- list on-line documentation
Isam	- generate index for pseudo-indexed-sequential access
Kill	- kill a running process
Kwic	- make keyword in context index
Lam	- laminate files
Lcnt	- line count
Ld	- loader
Ll	- print line lengths
Lpr	- queue file to printer
Ls	- list contents of directory
Macro	- process macro definitions
Man	- run off section of users manual
Mcol	- multicolumn formatting
MkDir	- create directories
Mv	- move (or rename) a file
Number	- number lines
Os	- convert backspaces into multiple lines for "printers"
Pack	- pack words into columns
Pl	- print specified lines/pages in a file
Pr	- paginate files to standard output
Printf	- justify fields of data in fixed-width fields
Prlabl	- format labels for printing
Ps	- list process status information
Pstat	- determine status of process
Pwd	- print working directory name on standard output
Rar	- rearrange archive
Ratfor	- RatFor preprocessor
Rc	- RatFor compiler
Resume	- resume a suspended process
Rev	- reverse lines
Rm	- remove files
Ruler	- display ruler on terminal screen
Sched	- a way to repetitively invoke a command
Sedit	- stream editor
Send	- send a message to another user's terminal
Sepfor	- Split FORTRAN programs into multiple files
Sh	- shell (command line interpreter)
Sleep	- cause process to suspend itself for a period of time
Sort	- sort and/or merge text files
Spell	- find spelling errors
Split	- split a file into pieces
Suspnd	- suspend a running process
Tail	- print last lines of a file
Tee	- copy input to standard output and named files
Timer	- time execution of a process
Tr	- transliterate characters
Tsort	- topologically sort symbols
Ttt	- 3-dimensional tic tac toe

Txtrpl	- perform generalized text replacement
Ul	- convert backspaces into multiple lines for "terminals"
Uniq	- strip adjacent repeated lines from a file
Unrot	- unrotate lines rotated by kwic
Wc	- count lines, words, and characters in files
Wcnt	- (character) word count
Whereis	- locate file in tree based on partial pathname
Who	- show who is on the system
Xch	- extended change utility
Xfind	- entended find utility
Xref	- make a cross reference of symbols

Formerly released mail utilities

Mail	- utility for sending mail to local users
Msg	- utility for manipulating message files
Msplit	- utility for salvaging message files
Postmn	- report the presence of mail
Resolve	- resolve mail system user names
Sndmsg	- utility for sending mail to other users
Users	- list valid mail users

SIG Tape Information on the Distribution

These tools are normally distributed on the SIG Tape as a BACKUP container with the name SWTOOLS.BCK The directories in the container file have the following significance:

[...DISTN]	All of the files necessary to build this release of the Tools on VMS. Note that it is now necessary for you to build the images from the source files in this directory. Images and objects are NO LONGER distributed. The system has successfully built on all versions of VMS >= 3.0.
[...MSGSYS]	The distribution of the Software Tools Distributed Mail System.
[...OLDMSG]	The TCS archives for the previously released mail utilities.
[...SRC]	The source for the portable VOS utilities.
[...VMS]	The source files for the VMS-specific tools and primitive functions.

On Disk Structure of the Tools VOS

The tools system uses 6 directories which may be scattered over one or more of your disks, with an optional seventh directory at the discretion of the site management. Each known directory is defined as a system logical name; each logical name is of the form ddnn:[dir...] - i.e. a device AND directory specification.

st_bin This defines '~bin', the directory in which the distributed images are built and kept. This directory should have the protection [rwe,rwe,re,re].

st_usr Site-specific tools, scripts and other known files should be kept here (~usr). The protection should be [rwe,rwe,re,re].

st_tmp The scratch files created by the tools are kept here (~tmp). As such, the directory must have the protection [rwe,rwe,rwe,rwe]. In addition, all users of the tools must have a quota on the disk which st_tmp points to, if quotas are enabled on that disk.

st_lpr The files formatted by the 'lpr' tool which are queued to the print symbiont are kept here (~lpr). The protections and quota considerations are the same as for ~tmp.

st_msg The known files for the mail system are kept here (~msg). The protection should be [rwe,rwe,re,re].

st_man The archives and indices used by the 'man' and 'intro' utilities are kept here (~man). The protection should be [rwe,rwe,re,re].

st_src (Optional) The source files for the tools modified locally should be placed here (~src).

In addition, the VOS requires two other system logical names to run:

st_node - the name of your node in a network; if you are not a member of a net, pick one that appeals to you.

st_timezone - the three character mnemonic for the timezone in which the machine is situated. Only the first character is used, as routines exist in the library to determine the state of standard/daylight time.

Two other logical names can be defined at the discretion of site management:

sys_tools - This should be defined as the same value as st_bin. It is only for compatibility with previous releases.

st_new_versions - If this is defined to be the value "YES", then the tools will create a new version of a file when writing a file. This feature has just been added, so there may be some complications with its use. The logical name can be defined in any of the three name tables and have the desired effect. As such, it is an individual option to define it at LBL.

Runtime requirements

The system logical names described above.

The file st_bin:tooldef.com defines the tools as foreign symbols so that they can be invoked from DCL. Invocation of the command file from a system login file guarantees the symbol definitions for the tools for all users when they log in. Alternatively, interested users may invoke @st_bin:tooldef in their individual login.com files.

Several of the images need to be installed with enhanced privilege to provide full functionality to all users. They are:

* ps (GROUP,WORLD) - lists valuable information on processes in the system.

* who (GROUP,WORLD) - lists who is logged into the system, and other info.

* send (OPER) - inter-terminal write facility that is not specific to any particular type of terminal.

* sh (DETACH,CMEXEC) - the DETACH privilege is required by the shell to permit the user to spawn background processes. If this feature is not supported locally, then do not install with the privilege. The CMEXEC privilege permits the shell to redefine the process logical name SYS$DISK at supervisor mode when performing a ‘cd’ command, such that the device assignment remains when leaving the shell. This is done by changing mode to EXEC, redefining the logical name at supervisor mode, and returning to USER mode.

* esh (DETACH,CMEXEC) - same as for sh.

* hsh (DETACH,CMEXEC) - same as for sh.

In addition, if ‘ed’ or ‘e’ are heavily used on your system, it is suggested that they be installed /SHARED/OPEN/HEADER_RESIDENT.

In the same vein, if the tools have been built with the shared global image, RLIBSHARE.EXE, it should be installed /SHARED/OPEN to facilitate global sharing of the tools runtime library. On V4.x systems, this name should be defined in EXEC mode. If you have problems with the tools or installed images, then try this: (1) Move a copy of RLIBSHARE.EXE from ~bin to the VMS directory SYS$SHARE. (2) Change the install procedure for the tools so that RLIBSHARE is installed from SYS$SHARE rather than ST_BIN:.

The file st_bin:tools.ins is a DCL command file which causes the above eight images to be installed with the above privileges, and can be invoked during system startup. The file st_bin:tools.rem may be used to deinstall these images during update. For V3.x and earlier systems, these files are named toolsv3.ins and toolsv3.rem respectively.

While the system is being built during TOOLGEN, the file st_bin:sysuaf.mod is generated, which is a DCL command file to cause authorize to modify all accounts on your system to reflect the suggested quota values to effectively use the tools. The suggested values are:

PRCLM	10
BYTLM	30000
FILLM	75
TQELM	40
PGFLQUOTA	16384

These values typically permit a user to have up to 5 processes active on his behalf. The most common problem incurred if the quotas are insufficient is the error message

"Cannot spawn process"

when attempting to invoke images from one of the shells. It is a good idea to peruse sysuaf.mod and remove accounts from it which do not need to be modified.

Source File Structure

The source code for ‘tool’ is contained in a file [...SRC]tool.tcs (if the tool is portable across operating systems) or [...VMS]tool.tcs (if it is an VMS-specific tool). This TCS source file contains an edit history of all changes made to the source. The output of the ‘get’ utility operating on a ‘.tcs’ file results in a file (tool.w) which is all of the environment necessary to rebuild the tool, provided that the VOS is operational. The tool.w file is an archive containing:

1. All of the files "included" by the ratfor source code.
2. The ratfor source file, tool.r.
3. The format input for the manual entry, tool.fmt.
4. And optionally, any extra definition files needed to build alternate versions of the tool (eg. sh => hsh).

As an example, suppose that you wish to change the subroutine "module" in "tool". The suggested scenario is as follows:

```
$ !Fetch the file tool.tcs from the appropriate directory in the container
$ !file on tape into st_src
$ hsh
% get ~src/tool.tcs tool.w
% ar xv tool.w
% ar xv tool.r module
% ed module
(make changes and write file)
% ar uv tool.r module
% rc -v tool.r
% (test out new tool.  repeat last three steps until satisfied.)
% ed tool.fmt
(modify writeup to reflect changes)
% ar uv tool.w tool.r tool.fmt
% cp tool.exe ~usr/tool.exe
% delta tool.w ~src/tool.tcs
(Identify in the comments the reason for the changes,
and which modules changed.)
% format tool.fmt >tool
% ar uv ~man/s1 tool
% asam <~man/s1 | sort >~man/i1
```

Placing tool.exe in ~usr causes the shell to find your modified version of "tool" rather than the distributed one. The last two commands above cause the manual entry for ‘tool’ to correctly correspond to the utility itself.

Source for Primitive and Library Functions

The source archive for the primitive and library functions may be found on [...VMS]rlib.w. This archive consists of several modules:

1. prim.m - an archive of macro files which are written in assembler and used by one or more tools. These routines are VMS-specific.

2. lib.m - assembler versions of portable ratfor routines which are used by one or more tools.

3. prim.r - archive of ratfor source routines which are VMS-specific and used by one or more tools.

4. lib.r - an archive of ratfor archives of portable library routines.

5. other files included by ratfor when processing prim.r.

To assemble any of the modules in prim.m or lib.m, it is necessary to extract the module(s) and assemble them individually

```
% ar xv prim.m directory.mar; mac/nolist directory
```

To modify one of the routines in prim.r, simply extract it using the archiver, edit it up, update the archive, and recompile via

```
% ar xv prim.r dscbld; ed dscbld; ar uv prim.r dscbld
% rc -cv prim.r
```

To modify one of the routines in lib.r, it is necessary to perform two extractions and two updates, as in

```
% ar xv lib.r arsubs.r
% ar xv arsubs.r aopen
% ed aopen
% ar uv arsubs.r aopen
% rc -cv arsubs.r
% ar uv lib.r arsubs.r
```

Of course, after generating new object modules for modified routines, it is necessary to make a system-specific version of st_bin:rlib.olb in st_usr, and to replace the object module in st_usr:rlib.olb. It is also a good idea to avoid replacing the modified modules in the archives until you are sure that they work. Writeups on all primitive routines which are to be visible to programmers may be found using the 'man' command on section 2 of the manual. Writeups for library routines are in section 3.

Manual entry structure

In order to simplify the generation of manual entries for utilities and functions, a set of 'format' macros are defined in the file '~bin/manhdr'. For the correct working of the 'intro' utility, it is necessary that the first three lines of any site-dependent writeups consist of

```
.so ~bin/manhdr
.hd <name> (<section>) (<date>)
one line description of the tool or function
```

where <name> is replaced by the name of the function or tool, <section> is the section of the manual this entry is for and <date> is the date the document was created. The .hd macro guarantees that the margins are correct, the header line on the manual pages is consistent with the software tools standard, and that

```
NAME
     name - one line description of the tool or function
```

appears in the writeup. This particular landmark is used by the intro utility to list the one-liners for the known entries in each section. The best method is to peruse the macros in ~bin/manhdr and to look at some of the writeups supplied with the system.

Legal file specifications for the tools

The following lists legal VMS file specs for the tools and valid tools pathname equivalents:

DEC format	Path format
file.typ.ver	file.typ.ver
[dir]file.typ.ver	/dir/file.typ.ver
[dir.sub...]file.typ.ver	/dir/sub/.../file.typ.ver
[.sub]file.typ.ver	sub/file.typ.ver
[-.sub]file.typ.ver	\sub/file.typ.ver
ddnn:[dir]file.typ.ver	/ddnn/dir/file.typ.ver
host::ddnn:[dir]file.typ.verp	/@host/ddnn/dir/file.typ.ver
?	~name/file.typ.ver
?	~/file.typ.ver

In all cases, the pathname equivalent consists of replacing the many and varied VMS delimiters by slashes, which is typically a lower-case character on all terminals and is normally easy to strike using the right pinky. In addition, the backslash (\) is used to go up in the directory tree, equivalent to DEC's [-] construct. The ~name capability is available for the seven known directories of the tools system, ~bin, ~usr, ~tmp, ~lpr, ~msg, ~man and ~src. They permit one to write portable scripts for utilities across different operating systems. Also, ~user, where 'user' is the login name of a user on the system maps onto that user's home directory.

The ~/ is shorthand for the user's home directory.

In utilities which manipulate directories, all of the above formats are valid when the file.typ.ver trailer is removed.

Changes for the Spring 1986 Release

Modified Utilities

* ‘addr’ has been modified to reflect V4 changes in the SYSUAF.DAT file. The code has also been conditionalized for V3.x or V4.x with the addition of VMSV3 in ratdef. The correct ratdef is selected during the toolgen process.

* ‘banner’ now includes the big character file.

* ‘ps’ has been modified to reflect changes made in valid directory names (with multiple] and [allowed). The Term field has also been widened for the display of longer terminal device names (such as virtual terminals).

* ‘sh’ has been modified many times to remove bugs and add features. Refer to changes.86 for more specific information.

* ‘who’ has been modified to reflect changes made in valid directory names (with multiple] and [allowed). The Term field has also been widened for the diplay of longer terminal device names (such as virtual terminals).

Changes for the Spring 1986 Release

New Utilities

* ‘cron’ executes commands at specified dates and times according to the instructions in the file ~usr/crontab. Since CRON never exits, it should only be executed once, usually when the system is booted.

```
 __  __                         _   ____
|  \/  | __ _ _ __  _   _  __ _| | |  _ \ __ _  __ _  ___  ___
| |\/| |/ _` | '_ \| | | |/ _` | | | |_) / _` |/ _` |/ _ \/ __|
| |  | | (_| | | | | |_| | (_| | | |  __/ (_| | (_| |  __/\__ \
|_|  |_|\__,_|_| |_|\__,_|\__,_|_| |_|   \__,_|\__, |\___||___/
                                               |___/
```

The Software Tools on-line documentation is divided into several sections. The standard manual sections contain the following information:

1 Writeups on the utilities available in the system (e.g. ed)
2 Writeups on the virtual machine system calls available to ratfor programmers.
3 Writeups on library routines available to ratfor programmers.
4 Primers on some of the more heavily used utilities.

In addition, other site-dependent manual sections may be maintained by your system manager.

Three utilities are currently available for perusing the on-line documentation: intro, apropos and man. Specific information can be obtained on each of these utilities by performing a command of the form:

```
man NAME
```

where NAME is replaced by intro, apropos or man.

In addition, some of the more heavily used forms of these commands are listed below.

```
man -s
```
List all available manual sections.

```
man -s1
```
List all available entries in manual section 1.

```
man ed
```
List the manual entry on 'ed'. If entries exist in more than one section, only the first is displayed, with a note concerning the other entries displayed following the first entry.

```
man -a -s2
```
Display all entries in section 2.

```
man -a
```
Display all entries for all sections (the entire manual).

```
intro -s2
```
Display a one-line synopsis of all entries in section 2.

```
apropos mail
```
Display a one-line synopsis of all entries in the manual that match the pattern 'mail'.

```
 ____            _   _               _ 
/ ___|  ___  ___| |_(_) ___  _ __   / |
\___ \ / _ \/ __| __| |/ _ \| '_ \  | |  _____ 
 ___) |  __/ (__| |_| | (_) | | | | | | |_____|
|____/ \___|\___|\__|_|\___/|_| |_| |_|

 _   _ _   _ _ _ _   _           
| | | | |_(_) (_) |_(_) ___  ___ 
| | | | __| | | | __| |/ _ \/ __|
| |_| | |_| | | | |_| |  __/\__ \
 \___/ \__|_|_|_|\__|_|\___||___/
```

NAME
    Intro - list on-line documentation

SYNOPSIS
    intro [-s<section]

DESCRIPTION
    Intro lists a short synopsis of each manual entry which is available for the specified section (section 1 is the default). Valid section names are described in the writeup for 'man'. These documents can then be accessed via the tool 'man'.

FILES
    Intro accesses the archive file containing the user documentation.

SEE ALSO
    man; the Unix tool 'help'

DIAGNOSTICS
    none

AUTHORS
    Joe Sventek

BUGS/DEFICIENCIES

NAME
Acat - concatenate nested archive entries on standard output

SYNOPSIS
```
acat archive`module[`module...] ...
```

DESCRIPTION
'acat' performs the equivalent function to 'cat' on archive files created by the 'ar' utility. The true power of 'acat' lies in its ability to extract the modules from within nested archives. Some examples may help clarify its use.

Suppose the file arch1 consists of the modules mod1a, mod1b and mod1c. In addition, mod1c is itself an archive, consisting of modules mod2a and mod2b. The command line

```
% acat arch1`mod1a
```

is equivalent to the 'ar' command

```
% ar p arch1 mod1a
```

More importantly, if the user desires to see mod2a in mod1c in arch1, the command

```
%acat arch1`mod1a`mod2a
```

will do the trick.

FILES

SEE ALSO
ar - archive file maintainer
cat - concatenate files

DIAGNOSTICS

AUTHORS
Joe Sventek

BUGS/DEFICIENCIES

NAME

Admin - administer TCS file.

SYNOPSIS

```
admin -ifile file.tcs
```

DESCRIPTION

Admin -i enters a text file into the TCS system for the first time. File is the source file to be entered into the system. Local convention is to use the name "file.tcs" for files maintained by TCS.

The file is tagged as Version #1.1 and the user is prompted for initial comments concerning the development of the file.

The date, time and user ID are recorded in the statistics portion of the file.

FILES

A scratch file is used while creating the output file and moved upon completion of input.

SEE ALSO

delta, get

DIAGNOSTICS

usage: admin -ifile file.tcs
: Correct calling format is provided when called without arguments.

- flag missing
: Incorrect calling procedure.

-i... filename missing
: The input filename is expected to be immediately adjacent to the -i flag. (no white-space)

Invalid flag
: -i is the only valid flag at present.

AUTHORS

Neil Groundwater at ADI.

BUGS/DEFICIENCIES

NAME
    Alist - generate paginated listing of source archive

SYNOPSIS
    alist [file] ...

DESCRIPTION
    'alist' generates a paginated listing of archive files. A table of contents with the relative page number in the listing is displayed first, with each element of the archive file starting on a new page. The second page of the listing contains a sorted index of entries, with the starting page number. If no files are specified, the standard input is read. 'alist' considers each line which starts with the string "#-h-" to be the beginning of a new entry, so that nested archives will be handled reasonably. The listing is displayed on standard output, and may be piped into lpr to queue to the printer.

FILES

SEE ALSO
    pr - print files

DIAGNOSTICS

AUTHORS
    Joe Sventek

BUGS/DEFICIENCIES

NAME

Apropos - locate manual entries matching user-specified patterns

SYNOPSIS

apropos pattern [pattern] ...

DESCRIPTION

apropos searches the on-line documentation system for entries which match the regular expression patterns supplied in the command line. For each entry so found, a one-line synopsis of the entry (similar to the one displayed by the tool ‘intro’) is displayed on standard output, with the section where the entry may be found. The ‘man’ utility can then be used to retrieve more information on the topic.

EXAMPLES

To get a list of all entries having anything to do with the mail system, type the following command:

```
                apropos mail
```

FILES

Accesses the known files for each section in the ˜man directory.

SEE ALSO

The tools ’intro’ and ’man’; the Unix command ’man’

DIAGNOSTICS

AUTHORS

Joe Sventek.

NAME
    Ar - archive file maintainer

SYNOPSIS
    ar {dpstux}[v/1] arcname [file] ...

DESCRIPTION
    Ar collects sets of arbitrary files into one big file and maintains that file as an 'archive'. Files can be extracted from the archive, new ones can be added, old ones can be deleted or replaced by updated versions, and data about the contents can be listed.

    If a minus sign ('-') is given as a file name, further file names are read from the standard input, one file name per line.

    Files that are to be added to an archive must exist as files with the name given. Files that are extracted from an archive will be put onto files with the name given. Files that are added to archives can, of course, be archive files themselves. There is no (theoretical) limit to the number of files that can be nested this way. Thus Ar provides the utility necessary to maintain tree-structured file directories.

    Ar is invoked by the command line

```
          Ar command archname [optional filenames]
```

    where 'command' is any one of 'uxtpds', optionally concatenated with 'v' or '1', specifying what operation to perform on the archive file named 'archname'. The possible commands are:

    u - Update named archive by replacing existing files or adding new ones at end. If the 'v' option is used, file names will be printed on the standard output as files are written to the new archived file.

    x - Extract named files from archive. Put onto file of the same name. If the 'v' option is added, file names will be printed on the standard output as files are extracted.

    d - Delete named files from archive. If the 'v' option is used, file names will be printed on the standard output as they are deleted from the archive.

p - Print named files on standard output. Using the 'v' option will cause the file name to precede the file.

t - Print table of archive contents. Normally, the table will contain only the file name. If the 'v' option is used, the table will also contain the file's length, type, and date and time of last change. By default, if the standard output is a terminal, ar will pack five names per line in the non-verbose mode. If the optional '1' option is used, the output is force to single column, which is the default is standard output is not a terminal. For example,

```
ar t archive
```

might generate the following output:

```
a               b               c               d
```

whereas

```
ar t1 archive
```

would generate

```
a
b
c
d
```

s - Salvage. This command may be used to recover a damaged archive whose character counts do not reflect the correct number of characters in the file. The 's' command extracts all files from the archive, ignoring characters counts, date and time stamps, etc. on the archive header lines; it simply uses '#-h-, which begins each archive member, and the file name which follows it. The files are then replaced in the archive, with corrected character counts. Thus, the 's' flag is useful for salvaging the contents of 'alien' archive files and for saving damaged archives. It does not work on nested archives (i.e. archives within archives).

v - Verbose. This command may be concatenated to any of the above commands, and will cause the archiver to print additional information, generally file names, on the standard output. Its specific action for each command has already been described.

The optional filenames in the command line specify individual files that may participate in the action. If no files are named, the action is done on ALL files in the archive, but if any files are explicitly named, they are the ONLY ones that take part in the action. (The 'd' command is an exception--files may be deleted only by specifying their names.)

FILES

A file 'arctemp' is created and subsequently deleted for each run.

SEE ALSO

The Unix commands 'ar' and 'ls' in the Unix manual
'rar' - rearrange archive

DIAGNOSTICS

archive not in proper format

The basic problem is that archive didn't find a header line where one was expected. Typical reasons include misspelling the file name, using an existing file (not in archive format) on a creation run, and referencing an archive file that has been modified directly (say with the editor).

delete by name only

For user protection, files are allowed to be deleted from an archive only by specifying each file name.

duplicate file name

A file was listed more than once when calling the archiver

fatal errors-archive not altered

This message is generated whenever one or more of the other errors have been detected. An archive is never altered unless EVERYTHING has run properly.

too many file names

At the present the user may call the archiver with no more than 25 files at a time.

usage: ar [dptuxsv] arcname [files]

The command line passed to the archiver is in error. Possibly the command is wrong or the archived file name has not been given.

'filename': can't add
: The file specified by 'filename' doesn't exist or can't be opened (e. g. is locked).

'filename': can't create
: The archiver could not generate a local file by the name of 'filename'. Probably the archiver's internal file buffer space has been exceeded.

'filename': not in archive
: The archiver could not locate the file specified by 'filename' in the archived file.

AUTHORS

Original code from Kernighan and Plauger's 'Software Tools', with modifications by Debbie Scherrer.

BUGS/DEFICIENCIES

On some systems only text files can be archived.

When the update and print commands are used, the files are updated or printed in the order they appear on the archived file, NOT the order listed on the command line.

The 's' salvage command works only on unnested archives.

The Unix archiver allows files to be positioned in the archive, rather than simply added at the end as Ar does. This is done by adding the following commands:

m - Move specified files to end of archive

ma posname - Move specified files to position after file 'posname'

mb posname - Move specified files to position before file 'posname'

r - Replace specified files and place at end of archive

ra posname - Replace files and place after file 'posname'

rb posname - Replace files and place before file 'posname'

There are some discrepancies between the Unix version of Ar and this version. Unix uses 'r'--replace instead of 'u'--update. Unix also requires the user to specify an additional command 'n' when creating a new archive.

NAME
    Args - use standard input as arguments for command

SYNOPSIS
    args [-v] tool [arguments]

DESCRIPTION
    Args reads the standard input file and concatenates the words found there onto the arguments passed to it. It then spawns the tool "tool" with those arguments. The first argument to Args which does not start with a "-" is taken to be the name of the tool to be invoked. Args uses the same search path as the shell, and if "tool" is a script file, a copy of the shell will be spawned reading that file for its commands. The optional -v argument causes Args to display the final command line on ERROUT before spawning the sub-process.

    The most common use of Args is as a form of argument explosion, as in the following example:

        Suppose you wish to delete all files which have the string "tst" somewhere in the filename. This may be accomplished with the following shell command line:

```
% ls tst | args rm -v
```

        All of the files matching the pattern "tst" will be fed to Args, which will concatenate the names onto Rm's command line. Rm will then be spawned, and will print the name of each file as it is deleted.

    If the information found on standard input is so voluminous as to cause the argument string to be too large, the command line is displayed on ERROUT and the process is NOT spawned.

FILES
    none

SEE ALSO
    sh - command line interpreter (for search path rules)

AUTHORS
    Joe Sventek

BUGS/DEFICIENCIES

NAME

Asam - generate index for archive file

SYNOPSIS

```
asam <input_archive
```

DESCRIPTION

'asam' generates the same type of index as 'isam', with the exception that index lines are generated only for the archive header lines. The generated index appears on standard output, and may be sorted or whatever necessary for the application. The primary key output in the index line is the name of the module. Unlike 'isam', there are no switches for output control in the generated index lines. The module name is output, followed by a blank character, followed by the formatted linepointer.

'asam' is used specifically to generate the indices for the manual sections found in ~man.

FILES

SEE ALSO

isam - generate index for indexed-sequential access

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME

Asplit - salvage garbaged archive files

SYNOPSIS

asplit [-tstring] [-v]

DESCRIPTION

asplit reads the standard input file, looking for lines beginning with the archive header flag (#-h-). Upon locating such a line, the next word after the header is used to generate a file name, and all lines read up to the next pseudo-header line are written onto that file. When generating the file name, only the characters found before a left parenthesis are used, if one is found. If the -t switch is used, the string appended to the -t is appended to each file name before the file is created, thus permitting a fixed tag string to be formatted into the file name. If the -v option is specified, the name of each file is reported on ERROUT as it is opened. Any lines found at the beginning of the file before the first pseudo-header line is copied to standard output.

asplit is commonly used to salvage an archive which has been garbaged, or to take a monster fortran source program file and break it up into subroutines. A script file (breakup) may be found on the tools binary directory which will cause each subprogram of the form "subroutine snarf" or "... function snarf" to be placed on a file of the name "snarf.qq". The only side effect of this transformation is that the source will be in lower case, and may be remedied by modifying the file breakup.

FILES

none

SEE ALSO

ar - file archiver: the -s switch does essentially the same thing as asplit, except that it tries to rebuild the source file as a new archive, which does not always work in pathological cases.
sepfor - split FORTRAN programs into multiple files

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME

Axref - cross reference symbols in archive files

SYNOPSIS

axref [-fr] [file] ...

DESCRIPTION

'axref' produces a cross-reference list of the symbols found in each of the named files on the standard output. Each symbol is listed followed by the numbers of the lines in which it appears. If no files are specified, of the file "-" is specified, 'axref' reads the standard input.

'axref' differs from 'xref' in that it generates a separate cross-reference list for each module found within an archive. Module boundaries are defined to be those lines which start with the string "#-h-", as generated by the file archiver. Each module is preceded with the label

file/module_name:

on the standard output.

A symbol is defined as a string of letters, digits and underlines that begins with a letter. Symbols exceeding an internal limit are truncated. This limit is determined by the MAXTOK definition in the source code, and is currently set to 15.

By default, 'axref' differentiates between upper- and lower-case letters. The '-f' option causes all letters within symbols to be folded to a single case.

Normally, the line numbers specified in the symbol table are relative to the current file being processed. The '-r' option causes the line numbers specified to be relative to the start of the current archive module.

FILES

SEE ALSO

xref - make a cross reference of symbols

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME

Banner - generate large banner lines

SYNOPSIS

banner [string]

DESCRIPTION

'banner' formats the specified text strings into large banner lines on standard output. If a command argument is specified, then that string is output; otherwise, standard input is read, with each line being displayed on standard output, until an EOF is detected. Each character is printed in a 7 x 7 window, with the character occupying the central 5 x 5 portion. The file '~bin/bigchar' can be consulted for the format of the character file.

FILES

~bin/bigchar

SEE ALSO

DIAGNOSTICS

If a character is detected which has no correspondence in the character file, a blank is displayed.

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME

BarGraph - draw a 0-100% bargraph of integer data

SYNOPSIS

```
bargraph -[cfFhmrRsw] file ...
```

DESCRIPTION

BarGraph draws a simple graph of integer data scaled to 0-100%. Each line of input is expected to contain two fields, a label and a number, both in ASCII. The label and its associated numeric value should be separated by one or more BLANKs or a TAB. There are several options for controlling the appearance of the graph:

-c<c>	use <c> as the character for plotting the ordinate instead of ``|'' (the default)
-f<c>	use <c> to fill the area under the bar
-F<c>	use <c> to fill the area over the bar
-h	output column headers
-m[c]	display the mean of the data, using ``c'' if specified instead of ``.'' (the default)
-r[c]	display ruling every 10% under the bar, using ``c'' if specified instead of ``:'' (the default)
-R[c]	display ruling every 10% over the bar, using ``c'' if specified instead of ``:'' (the default)
-s	plot the running sum of the data values rather than the values themselves
-w	make the graph 132 columns wide rather than 80

EXAMPLES

```
bargraph -h -f* -w file
```

would make a 132-column wide graph of the data in ``file'', with column headers and ``*'' characters filling under the bar.

```
d "-f16n 9c" | bargraph "-f|"
```

would graph the sizes of all files in the current directory using ``|'' as the fill character below the bar. Note that it is necessary to put quotes around character specification options when specifying characters that are special to the shell (the "-f|" in the above example).

FILES

none

SEE ALSO

DIAGNOSTICS

? Too much data: increase TABLE_SIZE

AUTHORS

Dave Martin (Hughes Aircraft)

BUGS/DEFICIENCIES

There is an upper limit to the number of points which may be graphed. This limit may be raised by increasing the value of TABLE_SIZE. Numeric values are currently limited to 7 digits or less.

NAME
Box - draw boxes around block structure of RatFor or C programs

SYNOPSIS
```
box [-e] [-d{device}] [-] file ...
```

DESCRIPTION
Box draws boxes around statement groups (beginning with "{" and ending with "}") to make them more legible. It is designed to be used as a pretty-printer for RatFor or C code that is indented as follows:

```
        level 0
        {
level 1
{
  level 2
}
level 1
        }
        level 0
```

For this input, box generates:

```
level 0
+-------------+
| level 1     |
| +---------+ |
| | level 2 | |
| +---------+ |
| level 1     |
+-------------+
level 0
```

The alignment of the "{" and "}" characters and the indentation of 2 spaces per level are required for proper operation. If TABs are present in the input, they are replaced with blanks, on the assumption of 8 spaces per TAB. The "-d" option takes advantage of the line-drawing character sets on certain devices; currently the DEC vt100 and the Heath h19 are supported.

EXAMPLES
```
box -dvt100 myfile
```

box the file "myfile" to STDOUT (assumed to be a vt100)

AUTHORS
Dave Martin (Hughes Aircraft)

BUGS/DEFICIENCIES

NAME
    Cat - concatenate and print text files

SYNOPSIS
    cat [-v] [file] ...

DESCRIPTION
    'cat' reads each file in sequence and writes it on the standard output. Thus

```
     cat file
```

prints the file, and

```
     cat file1 file2 >file3
```

concatenates the first two files and places the result on the third.

If no argument or '-' is given, 'cat' reads the standard input.

If the '-v' option is specified, all control characters are displayed as '^C', where C is the character that must be typed with the CTRL key when entering the character. If any DEL(RUB) characters are found, they are displayed as '^?'. A dollar sign character ('$') is displayed at the end of each line to aid in location of trailing blanks in lines.

FILES
    none

SEE ALSO
    The "Software Tools" book, p. 77.
    The UNIX tools cat, PR, CP

DIAGNOSTICS
    A message is printed if a file cannot be opened; further processing is terminated.

AUTHORS
    Dennis Hall, Debbie Scherrer and Wen-Sue Gee.

BUGS/DEFICIENCIES
    Using the same file for output as well as input may cause strange results.

NAME

Ccnt - character count

SYNOPSIS

```
ccnt [file] ...
```

DESCRIPTION

ccnt counts characters in the named file(s). Newlines are counted as characters. If no file name or the file '-' is given, standard input will be read.

FILES

none

SEE ALSO

wcnt - count words
lcnt - count lines
the Unix command 'wc'

DIAGNOSTICS

A message is printed if an input file cannot be opened; further processing is terminated.

AUTHORS

Original from Kernighan and Plauger's 'Software Tools', with minor modifications by Debbie Scherrer.

BUGS/DEFICIENCIES

NAME

Ch - make changes in text files

SYNOPSIS

```
ch [-ax] [expression] ... fromexpr [toexpr]
```

DESCRIPTION

ch copies each line of the standard input to the standard output, globally substituting the text pattern "toexpr" for "fromexpr" on each line that satisfies matching criteria defined by the leading expressions "expression" and the switches. (A text pattern is a subset of a "regular expression"--see the "ed" writeup for a complete description.) Three possible courses of action are taken depending upon the number of text patterns(n) found in the command line:

```
n=1  The text pattern is assumed to be "fromexpr"  with  a  null
     "toexpr"; it is equivalent to the ed command
               g/fromexpr/s///g
n=2  The  first  text  pattern  is  "fromexpr",  the  second  is
     "toexpr"; it is equivalent to the ed command
               g/fromexpr/s//toexpr/g
n>=3 The (n-1)th pattern is "fromexpr", the nth is "toexpr"  and
     patterns  1...n-2  are  used  to  determine  the lines upon
     which to perform the substitution.   The  default  is  that
     any  line  which  matches  any  one  of  the  n-2  leading
     expressions are eligible for substitution.  If the -a  flag
     is  specified,  only  lines  which  match  all  n-2 leading
     expressions in any order are eligible.  If the -x  flag  is
     specified,   all   lines  which  don't  satisfy  the  above
     criteria are eligible. (See the writeup on  find  for  more
     information.)  In particular, if n=3,
               ch expr from to
     is equivalent to the ed command
               g/expr/s/from/to/g
               ch -x expr from to
     is equivalent to the ed command
               x/expr/s/from/to/g
```

The substitution string "toexpr" may be a string of replacement characters, null to effect a deletion, or it may include the special "ditto" character "&" to put back the "fromexpr" string and thus effect an insertion. It may also contain the expressions `$1' ... `$9', which cause the corresponding tagged pattern in the input to be inserted. If a deletion is desired with the multiple leading tag expressions, a "toexpr" of "" -i.e. quotes around an empty string may be used.

A text pattern consists of the following elements:

c	literal character
?	any character except newline
%	beginning of line
$	end of line (null string before newline)
[...]	character class (any one of these characters)
[!...]	negated character class (all but these characters)
{expr}	tagged pattern (referenced by $1 ... $9)
*	closure (zero or more occurrences of previous pattern)
+	anchored closure (one or more occurrences of previous pattern)
@c	escaped character (e.g., @%, @[, @*)

Any special meaning of characters in a text pattern is lost when escaped, inside [...], or for:

%	not at beginning
$	not at end
*	at beginning
+	at beginning

A character class consists of zero or more of the following elements, surrounded by [and]:

c	literal character
a-b	range of characters (digits, lower or upper case)
!	negated character class if at beginning
@c	escaped character (@! @- @ @])

Special meaning of characters in a character class is lost when escaped or for

!	not at beginning
-	at beginning or end

An escape sequence consists of the character @ followed by a single character:

@f	formfeed
@l	linefeed
@n	newline
@r	return
@t	tab
@OOO	the octal digit representation for an ASCII character for example, @001 for the ASCII character SOH
@c	c (including @)

For a complete description, see "Software Tools" pages 135-154. Care should be taken when using the characters % $ [ ] ! * + @ and any shell characters in the text pattern. It is often necessary to enclose the entire substitution pattern in quotes.

FILES
    none

SEE ALSO
    The UNIX tool GRES
    The tools find and ed
    xch - extended change utility

DIAGNOSTICS
    An error message is printed if the pattern given is illegal.

AUTHORS
    'CH' was originally implemented on BKY by Debbie Scherrer from Kernighan and Plauger's "Software Tools". Major modifications were performed by Joe Sventek.

BUGS/DEFICIENCIES
    A minus sign(dash[-]) may not start an expression.

NAME

Chmod - change mode (protection codes) of file

SYNOPSIS

```
chmod system owner group world file...
```

DESCRIPTION

Chmod allows you to change the protection bits on one or more files. The protection fields for system, owner, group and world are specified by groups of the following characters:

a	allow all access
r	allow read access
w	allow write access
e	allow execute access
d	allow delete access
n	allow no access

Each of the four fields must be present and in the proper order.

EXAMPLES

```
chmod re rwed re re prog1.exe prog2.exe

chmod a a r r text.fmt

chmod n rwed n n secret.txt
```

FILES

none

SEE ALSO

The UNIX command "chmod".

DIAGNOSTICS

? Can't change protection of file ``filename''.

AUTHORS

Dave Martin (Hughes Aircraft)

BUGS/DEFICIENCIES

If you deny yourself write access to a file you own you will have to resort to the DCL "set protection" command to regain it.

NAME
    Chown - change the ownership of file(s).

SYNOPSIS
    chown user file ...

DESCRIPTION
    chown makes "user" the owner of all listed files. "User" may be specified either as a username or a UIC ([ggg,mmm]).

FILES
    The mail system database "~msg/address" is used to resolve usernames into UICs.

SEE ALSO
    The UNIX command "chown".

DIAGNOSTICS
    A message is displayed if you don’t have the necessary privilege to change a file’s owner.

AUTHORS
    Dave Martin (Hughes Aircraft)

BUGS/DEFICIENCIES

NAME
    Cmp - compare two files

SYNOPSIS
    cmp file1 [file2]

DESCRIPTION
    file1 is compared line-by-line with file2. If file2 is not specified, standard input is used. If any lines differ, cmp announces the line number and prints each file's offending line.

FILES
    none

SEE ALSO
    comm
    The UNIX commands cmp, diff, and comm

DIAGNOSTICS
    If the end of one file is reached before the end of the other, a message is printed.

AUTHORS
    Acquired from "Software Tools" by Kernighan and Plauger, with minor modifications made by Debbie Scherrer.

BUGS/DEFICIENCIES
    If either file is binary, spurious results should be expected.

    Cmp cannot handle offset lines: line n of file1 is simply compared to line n of file2.

    Trailing blanks are significant, which will cause some lines to appear similar to the user which are actually different.

NAME

Comm - print lines common to two files

SYNOPSIS

comm [-123] file1 [file2]

DESCRIPTION

comm reads file1 and file2, which should be sorted, and produces a three column output: lines only in file1, lines only in file2, and lines in both files. The filename '-' means the standard input. If there is only one file argument, file2 refers to the standard input.

The optional arguments -1, -2, and -3 specify the printing of only the corresponding column. Thus "comm -3" prints only the lines common to both files, and "comm -12" prints lines which are in either file, but not in both. The default is -123.

FILES

none

SEE ALSO

cmp - compare two files
the Unix tool "diff"

DIAGNOSTICS

A message is printed if an input file cannot be opened.

AUTHORS

Debbie Scherrer

BUGS/DEFICIENCIES

The flags used by this tool are the reverse of those used by the Unix 'comm'. In Unix, the flags 1, 2, and 3 suppress printing of the corresponding column. Kernighan, on page 126 of 'Software Tools' suggests the version used above.

NAME
    Cp - copy files

SYNOPSIS
    cp [-v] from [to]

DESCRIPTION
    Cp duplicates file ``from'' into file ``to''. If the ``to'' argument is omitted, ``*'' is assumed. If the ``-v'' (verbose) option is specified, a confirming message is displayed as each file is copied.

EXAMPLES

```
            cp file.c file.bak
```

    would make a backup copy of ``file.c'' called ``file.bak''.

```
            cp ~usrlib/command.fmt
```

    would make a copy of ``~usrlib/command.fmt'' in the current directory keeping the same name.

```
            cp -v ~src/*.w /mt/*
```

    would make a backup copy of all files with an extension of ``.w'' in directory ``~src'' onto magnetic tape, confirming each file copied.

FILES
    none

IMPLEMENTATION
    Cp spawns the DCL ``copy'' command after converting the two arguments from pathnames to filespecs. If the ``-v'' option is specified, the DCL ``/log'' qualifier is added.

SEE ALSO
    mv -- move files
    The UNIX command ``cp''.

DIAGNOSTICS
    ? Can't spawn ``copy''.

AUTHORS
    Dave Martin (Hughes Aircraft)

BUGS/DEFICIENCIES
    DCL wildcards work; regular expressions don't.

NAME
    Cpress - compress input files

SYNOPSIS
    cpress [file] ...

DESCRIPTION
    cpress compresses runs of repeated characters in the input files. The output file can eventually be expanded with the tool 'expand'.

    If no input files are given, or the filename '-' appears, input will be from the standard input.

FILES
    none

SEE ALSO
    expand

DIAGNOSTICS
    A message is printed if an input file cannot be opened; further processing is terminated.

AUTHORS
    From Kernighan & Plauger's 'Software Tools', with modifications by Debbie Scherrer.

BUGS/DEFICIENCIES

NAME
    cron - clock daemon

SYNOPSIS
    ~bin/cron

DESCRIPTION
    CRON executes commands at specified dates and times according to the instructions in the file ~usr/crontab. Since CRON never exits, it should only be executed once, usually when the system is booted.

    Crontab consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns to specifiy:

minute	0-59
hour	0-23
day of the month	1-31
month of the year	1-12
day of the week	1-7 (1 => Monday)

    Each of these patterns may contain a number in the range above; two numbers separated by a minus meaning a range inclusive; a list of numbers separated by commas meaning any of the numbers; or an asterisk meaning all legal values. The sixth field is a string that is executed by the Shell at the specified times.

    CRONTAB is examined by CRON at periodic intervals, usually between 1 and 10 minutes.

EXAMPLES
```
                0,10,20,30,40,50 9-17 * * 1-5 command
```

    Execute command every 10 minutes from 9AM-5PM Monday-Friday.

```
                         23 50 * * 5 command
```

    Execute command at 10 minutes before midnight every Friday.

FILES
    ~usr/crontab
    ~usr/cron.log

AUTHORS
    Joe Sventek

NAME

Crt - copy files to terminal a screen at a time

SYNOPSIS

crt [-n] [file] ...

DESCRIPTION

crt is similar to 'cat' except that it prints only n lines (default 22) at a time. After each set of lines are printed, crt will wait for instructions from the user. Hitting a SPACE or RETURN will cause the next n lines to appear, hitting a 'q' (quit) will cause crt to skip over to the next input file (if any), and hitting an end-of-file character (^Z) will cause crt to stop immediately.

If no files are specified, or if the filename '-' is given, lines will be read from the standard input.

The flag -n may be given, where n specifies the number of lines desired at a time.

crt will stop at the end of each file (except the last), as well as after each n lines.

FILES

none

SEE ALSO

cat

DIAGNOSTICS

A message is printed if an input file cannot be opened; further processing is terminated.

AUTHORS

Debbie Scherrer; Modified to use RARE i/o by Dave Martin.

BUGS/DEFICIENCIES

NAME
    Crypt - crypt and decrypt standard input

SYNOPSIS
    crypt key

DESCRIPTION
    crypt encrypts characters on the standard input by using 'key'. The file can eventually be decrypted by running it back through crypt with the same key. Double encryption (encrypting a file with first one key and then another) is allowable, but on some systems the decryption must be done in the exact reverse order as encryption was done.

    The encryption algorithm used by 'crypt' is not a complicated one, so users requiring a great degree of protection should not rely on this tool.

FILES
    none

SEE ALSO

DIAGNOSTICS

AUTHORS
    Original from Kernighan & Plauger's 'Software Tools', with modifications by Debbie Scherrer. (NOTE: the original encryption algorithm has been altered slightly.)

BUGS/DEFICIENCIES
    On IAS and VMS systems, double encryption must be decrypted in the exact reverse order as the encryption.

NAME
    D - list contents of directory

SYNOPSIS
    d [-1dhnrtv] [-fstring] [pathname] ...

DESCRIPTION
    D lists information about each file argument. When no argument is given, the default directory is listed. The file arguments may include any of the legal regular expressions described in the man entry for the editor, with the added feature that the comparisons will be case insensitive. By default, the files are listed in the order in which they are found in the directory. There are seven options:

```
-1 force single column output to the terminal.  The default is
   multi-column output to the terminal, single to a disk file.
-d print only directory files found in this directory
-h print a header at the top of verbose listings
-n sort the directory by name
-v list in verbose format
-t sort by time modified (oldest first)
-r reverse the sense of the sort

-f use 'string' to specify the output format as follows:


      b  size of file in blocks (normally 512 characters)

      c  size of file in characters

      m  modification date and time (dd-mmm-yy hh:mm:ss)

      n  filename

      o  file owner's username

      p  protection codes (oooo|gggg|wwww)

      t  file type (asc|bin|dir)
```

The 'b', 'c', 'n' and 'o' options accept an integer prefix which specifies the field width to be used.

The verbose option formats its output as if you had specified "-f17n 9c t m p o" as a format string.

It is necessary to surround the string (including the '-f') with quotes if it contains any BLANKs or TABs.

EXAMPLES

The following command will cause all of the files which contain the string tst anywhere in the file name to be deleted:

```
% d tst | args rm
```

FILES

lstemp1, lstemp2

AUTHORS

Ls was written by Joe Sventek. The '-f' option was added by Dave Martin.

SEE ALSO

ed - text editor for description of regular expressions
args - argument exploder
ls - directory lister (with different default format)
fd - fast directory lister in sort order

NAME
Date - print the date

SYNOPSIS
date [-n]

DESCRIPTION
The current day of the week, date, time, and time zone are printed in the format:

```
day dd-mmm-yy hh:mm:ss zone
```

if the -n switch is used the date is output in the following format:

```
day mm/dd/yy hh:mm:ss zone
```

FILES
none

SEE ALSO
The Unix command 'date'

DIAGNOSTICS
none

AUTHORS
Debbie Scherrer

BUGS/DEFICIENCIES

NAME

Dc - desk calculator

SYNOPSIS

```
dc [file] ...
```

DESCRIPTION

dc evaluates integer expressions from the source files, one expression per input line. If no input files are given, or the filename '-' is specified, dc reads from the standard input.

Ordinarily dc operates on decimal integer arithmetic expressions, but the user may specify an input base and output base other than decimal.

Expressions may be simple arithmetic expressions or replacement expressions. The values of simple expressions are written on standard output when they are evaluated. Replacement expressions are used to hold temporary values, and are not automatically printed.

A simple expression is a normal arithmetic expression using numbers, variables, parentheses, and the following operators, listed in order of precedence:

```
     +  -          unary plus and negation operators.  These may
                  only appear at the start of a simple
                  expression or after a "("

     **            exponentiation

     *   /   %     multiply, divide, modulo (remainder)

     +   -         add, subtract

     == !=         relations - equals, not equal to,
     >  >=         greater than, greater than or equal to,
     <  <=         less than, less than or equal to
              (!=, ^=, ~= all treated as "not equal")

     !             unary logical not (also ~ and ^)

     |   &         logical or, and
```

The logical operators ! | & and the relational operators result in the values 1 for true and 0 for false.

A replacement expression is:

```
                    name = simple expression
```

where 'name' is a character string of (virtually) any length, starting with a letter and consisting of only letters and digits. (The characters a-f should not be considered letters when operating in hexadecimal mode.) Variables are automatically declared when they first appear to the left of an "=" sign, and they should not be used in a simple expression until they have been declared.

Radix Control

Radix control is available in 2 ways:

1) There are default radix values for both input and output which may be changed by setting the predefined variables 'ibase' (input base) and 'obase' (output base). (Radix 10 is always used to evaluate and/or print radix-defining expressions.) For example,

```
ibase = 2
obase = 16
```

would accept input in binary and print results in hexadecimal.

2) The radix of individual numbers may be explicitly given by following the number with an underscore character and then the desired radix. For example,

```
100_16
```

would specify the hex number 100 (256 in decimal).

EXAMPLES

```
10 + (-64 / 2**4)
```

would print the answer "6"

```
temp = 101_2
temp == 5
```

would print the answer "1" (true)

```
ibase = 16
obase = 2
1a + f
```

would print the answer "101001"

```
ibase = 16
numa = 100_10
numb = 100
numa + numb
```

would print the answer "356"

FILES
    none

SEE ALSO
    macro, the UNIX M4 macro package
    The UNIX tools dc and bc

DIAGNOSTICS
    arith evaluation stack overflow
        arithmetic expressions have been nested too deeply. The size of the stack is set by the MAXSTACK definition in the source code.

    number error
        an input number has a number/character bigger than the current radix

    expression error
        invalid arithmetic expression

AUTHORS
    Philip H. Scherrer (Stanford U.)

BUGS/DEFICIENCIES
    dc only works with integers

    The maximum value allowed depends on the host machine and is the largest Fortran integer

NAME
    Delta - make an TCS delta

SYNOPSIS
    delta revision history [newhistory]

DESCRIPTION
    Delta integrates the current "revision" of a file into its TCS "history" file or into a "newhistory" file. Differences between this version and the preceeding version are computed and the TCS file will be able to reproduce either version (or earlier versions) by means of the GET command.

    The user is requested to provide a reason-for-change when prompted by "History?". Multiple lines may be entered to describe changes and terminated by '.' on a line by itself.

FILES
    A scratch file is created during processing, then copied onto the "history". If a "newhistory" is given, the result will be moved there instead.

SEE ALSO
    admin, get

DIAGNOSTICS
    usage: delta revision history [newhistory]
        Correct calling format is provided when called without arguments.

    TCS Version Number corrupted.
    Unexpected EOF on history-info scan.
    Unexpected EOF on history-data scan.
        The TCS code seems to be present but garbled. Refer to a guru.

    Sudden death in input
        An end-of-file was detected while requesting the "reason for change".

    Revision file is empty
        Perhaps an incorrect filename was given.

    History file is empty
        The first formal version is entered by means of the ADMIN command.

    Files are too big to handle
        The DIFF algorithm table-size has been exceeded. Current version supports files of approximately

15000-lines.

Cannot locate TCS history file.
Unable to read filename specified as the history file.

Temp file error: (filename)
The tempoary file created during processing disappeared unexpectedly.

AUTHORS

An Algorithm for Differential File Comparison by J.W.Hunt and M.D.McIlroy (BTL Computing Science Technical Report #41). Original code by Wil Baden; converted from MORTRAN by Dave Murray. Modifications and conversion to BTL-SCCS style by Neil Groundwater at ADI. The Source Code Control System was introduced by Marc J. Rochkind in the December, 1975, IEEE Transactions on Software Engineering.

BUGS/DEFICIENCIES

File permissions are NOT manipluated to restrict users from disturbing the maintained files.

Version numbering ranges from 1.1 to 1.N where N is a very large number. Provision to increment the "primary" number upon demand is scheduled.

Branching capabilities are scheduled to be implemented.

NAME

Detab - convert tabs to spaces

SYNOPSIS

```
detab [<t1>...] [+<n>] [file] ...
```

DESCRIPTION

detab converts tab characters (control-i) to equivalent strings of blanks. Tab stops are indicated by <t1>... (default 8, 16, ...), while +<n> indicates tab stops every <n> columns. Thus the command

```
detab 5 21 +5
```

supplies blanks for tabs terminating at column positions 5, 21, 26, etc. If no files are specified, the standard input is read. An isolated minus sign also indicates the standard input.

SEE ALSO

entab
lpr

AUTHORS

Original from Kernighan & Plauger's 'Software Tools', with modifications by Dennis Hall and Debbie Scherrer.

BUGS/DEFICIENCIES

NAME
    Diff - isolate differences between files

SYNOPSIS
    diff [-{c|d|r|s|v}] old_file [new_file]

DESCRIPTION
    'Diff' compares the contents of two files and reports on the differences between them. The default behavior is to describe the insert, delete, and change operations that must be performed on 'old_file' to convert its contents into those of 'new_file'.

    The second file name argument is optional. If omitted, the standard input is read for the text of the 'new_file'.

    The options currently available are:

    -c Perform a simple line-by-line comparison. 'Diff' will compare successive lines of the input files; if any corresponding lines differ, or if one file is shorter than the other, 'diff' prints the message "different" and exits. If the files are the same, 'diff' produces no output. When the "-v" option (see below) is specified, 'diff' prints the lines that differ along with their line number in the input file, and notifies the user if one file is shorter than the other.

    -d List the "differences" between the two files, by highlighting the insertions, deletions, and changes that will convert 'old_file' into 'new_file'. This is the default option. If the "verbose" option "-v" (see below) is specified, unchanged text will also be listed.

    -r Insert text formatter requests to mark the 'new_file' with revision bars and deletion asterisks. This option is particularly useful for maintenance of large documents, like Software Tools reference manuals. (At present, only GT's version of 'format' can produce revision bars.)

    -s Output a "script" of commands for the text editor 'ed' that will convert 'old_file' into 'new_file'. This is handy for preparing updates to large programs or data files, since generally the volume of changes required will be much smaller than the new text in its entirety.

-v Make output "verbose." This option applies to the "-c" and "-d" options discussed above. If not selected, 'diff' produces "concise" output; if selected, 'diff' produces more verbiage.

'Diff' is based on the algorithm found in Heckel, P., "A Technique for Isolating Differences Between Files", _Comm. ACM_ 21, 4 (April 1978), 264-268.

EXAMPLES

To print the differences between two files on your terminal:

```
diff -cv file maybe_the_same_file
```
does a simple line-by-line comparison on the two files, printing lines which differ.
(Expects no missing or extra lines.)
Same as 'cmp file1 file2'.

```
diff -s old_version new_version | ed - old_version
```
make an ed script which changes 'old_version' into 'new_version'

```
diff -r old_manual.fmt new_manual.fmt | format
```
to mark changes in a document.
Useful only if your version of 'format' has this capability.

```
diff -s old new >>update_old_to_new
```
to keep a list of changes made to an original source file

DIAGNOSTICS

"<file>: can't open" if either `new_file' or `old_file' is not readable.

"Usage: diff . . ." for illegal options.

AUTHORS

Allen Akin and friends, Georgia Institute of Technology

BUGS/DEFICIENCIES

The algorithm used has one quirk: a line or a block of lines which is not unique within a file will be labeled as an insertion (deletion) if its immediately adjacent neighbors both above and below are labeled as insertions (deletions).

Fails on very large files (> 10000 lines on VMS).

NAME
E - extended version of "ed" with command editing & history

SYNOPSIS
e [-] [-pprompt] [-n] [-v] [file]

DESCRIPTION
e is an extended version of ed which uses virtual memory rather than a scratch file for its text storage. This makes it considerably faster than ed. In addition, command editing & history are supported; see the writeup on "esh" for more information.

Other commands and features which may not have found their way into ed:

1. There is a terse help command, invoked via ‘h’.

2. One can cause the current contents of the buffer to be roffed by issuing the "typeset" command via ‘t’. This causes format to be spawned, formatting the buffer contents to the terminal. The buffer contents are not affected. If more sophisticated use of format is necessary, or you deires to spawn something other than format, see the ed writeup for the ‘^’ command.

3. A command is available to see how much of the virtual memory array space has been used via ‘%’. If you exhaust the array space with many changes, simply writing the file followed by the enter command will cause garbage collection to occur.

For information on the other commands to e, consult the manual entry for ed.

FILES

AUTHORS
The extra features of e above those of ed are due to Dave Martin.

SEE ALSO
ed - text editor

BUGS/DEFICIENCIES

NAME
    Echo - echo command line arguments

SYNOPSIS
    echo [arg] ...

DESCRIPTION
    Echo writes its arguments in order as a line on the standard output file. It is useful for producing messages and diagnostics in command files.

FILES
    none

SEE ALSO
    The Unix command "echo"

DIAGNOSTICS
    none

AUTHORS
    Debbie Scherrer

NAME

Ed - line-oriented text editor

SYNOPSIS

```
ed [-] [-pstring] [-n] [-v] [file]
```

DESCRIPTION

Ed is a text editor. If the 'file' argument is given, the file is read into ed's buffer so that it can be edited and its name is remembered for possible future use. Ed operates on a copy of any file it is editing; changes made in the copy have no effect on the file until a w (write) command is given.

The optional '-' suppresses the printing of line counts by the e (edit), r (read), and w (write) commands.

The -p flag may be used to specify ed's prompt string. The default is ": ". If prompting is not desired, a bare -p in the command line will turn it off.

The -n flag indicates that you want to see line numbers prepended to each line of the buffer.

The -v flag indicates that each command is to be echoed on error output as it is executed.

Ed accepts commands from script files as well as a terminal. To do this, invoke ed and substitute the script file name for the standard input, as follows -

```
                ed [file] <script
```

Commands to ed have a simple and regular structure: zero , one, or two line addresses followed by a single character command, possibly followed by parameters to the command. The structure is:

```
       [line],[line]command <parameters>
```

The '[line]' specifies a line number or address in the buffer. Every command which requires addresses has default addresses, so the addresses can often be omitted.

Line addresses may be formed from the following components:

17	an integer number
.	the current line
$	the last line in the buffer
.+n	"n" lines past the current line

```
   .-n          "n" lines before the current line
   /<pattern>/  a forward context search
   \<pattern>\  a backward context search
```

Line numbers may be separated by commas or semicolons; a semicolon sets the current line to the previous address before the next address is interpreted. This feature can be used to determine the starting line for forward and backward context searches ("/" and "\").

REGULAR EXPRESSIONS

Ed includes some additional capabilities such as the ability to search for patterns that match classes of characters, that match patterns only at particular positions on a line, or that match text of indefinite length. These pattern-seaching capabilities include a class of patterns called regular expressions. Regular expressions are used in addresses to specify lines and in the s command to specify a portion of a line which is to be replaced. To be able to express these more general patterns, some special characters (called metacharacters) are used. The regular expressions allowed by ed are constructed as follows:

1. An ordinary character (not one of those discussed below) is a regular expression and matches that character.

2. A percent "%" at the beginning of a regular expression matches the empty string at the beginning of a line.

3. A dollar sign "$" at the end of a regular expression matches the null character at the end of a line.

4. A question mark "?" matches any character except a newline character.

5. A regular expression followed by an asterisk "*" matches any number of adjacent occurrences (including zero) of the regular expression it follows.

6. A regular expression followed by a plus "+" matches one or more adjacent occurrences of the regular expression it follows (anchored closure).

7. A string of characters enclosed in square brackets "[ ]" matches any character in the string but no others. If, however, the first character of the string is an exclamation point "!" the regular expression matches any character except the characters in the string (and the newline).

8. A string of regular expressions enclosed in braces "{}" is known as a tagged pattern, and can be referenced positionally as $1...$9 in the replacement side of a substitute command.

9. The concatenation of regular expressions is a regular expression which matches the concatenation of the strings matched by the components of the regular expression.

10. The null regular expression standing alone is equivalent to the last regular expression encountered.

If it is desired to use one of the regular expression metacharacters as an ordinary character, that character may be escaped by preceding it with an atsign "@".

COMMANDS

Following is a list of ed commands. Default addresses are shown in parentheses:

```
(.)a
<text>
.
```

The append command reads the given text and appends it after the addressed line. '.' is left on the last line input, if there were any, otherwise at the addressed line.

```
(.)b[+/./-][<screensize>]
```

The browse command is a shorthand command to print out a screenful of data. It has three basic forms, any of which may have a number("screensize") appended to it. The default screensize is 23. The b- form will print the screen of text preceding (and including) the addressed line; b. prints the screen centered on the addressed line; and b or b+ prints the current line and the screen after it. "." is left at the last line printed. If a screensize is specified, it becomes the default screensize for the rest of the editing session or until changed again.

```
(.,.)c
<text>
.
```

The change command deletes the addressed lines, then

```
     accepts  input  text   which  replaces these lines.  '.' is
     left at the last line input, if  there were any,  otherwise
     at the first line not deleted.

(.,.)d
     The  delete  command  deletes  the addressed lines from the
     buffer.  The line originally AFTER the  last  line  deleted
     becomes  the  current  line;  however, if the lines deleted
     were originally at the end, the new last line  becomes  the
     current line.

e filename
     The  edit  command causes the entire contents of the buffer
     to be deleted  and then the named file to be read in.   '.'
     is  set  to  the  last  line  of the buffer.  The number of
     lines  read  is  typed.  'Filename'  is   remembered   for
     possible  use  as  a  default file name in  a subsequent  r
     or  w  command.  If changes have been made to  the  current
     file  since  the  last  write command, you will be asked to
     repeat the edit command.

f filename
     The filename command prints the currently  remembered  file
     name.   If  'filename'  is  given, the currently remembered
     file name is  changed to 'filename'.

(1,$)g/regular expression/command
     In the global command, the given command  is  executed  for
     every   line  which  matches  the given regular expression.
     Multiple commands may be executed  by  placing  each  on  a
     preceding  line and terminated each command except the last
     with an atsign '@'.

h
     The help command causes a synopsis of the  commands  to  be
     displayed  on  standard  output.   If no help is available,
     that fact is noted on error output.

(.)i
<text>
.
     The insert  command  inserts  the  given  text  BEFORE  the
     addressed  line.  '.' is left at the last line input, or if
     there were none, at  the   addressed  line.   This  command
     differs  from  the   a   command  only  in the placement of
     text.

(.,.+1)j
     The join command joins the specified lines into  one  line.
     '.'  is  left  at the new line created by the join.  If the
```

join would result in a line longer than MAXLINE characters, an error is reported and no changes are made to the file. A trailing p or l may be given on the join command to cause the merged line to be printed or listed.

(.,.)k<address>
The kopy command copies the range of lines after the line specified by <address>. The last of the copied lines becomes the current line.

(.,.)l
The list command prints the addressed lines, expanding all ASCII characters with values between 1 and 31 (^A - ^_) as the appropriate two character digraph, ^(character). The end of line is also indicated by a '$'. '.' is left at the last line listed. The l command may be placed on the same line after any other command to cause listing of the last line affected by the command.

(.,.)m<address>
The move command repositions the addressed lines after the line specified by <address>. The last of the moved lines becomes the current line.

n[+/-/=][value]
This command manipulates the number register maintained by ed. A bare ‘n’ causes the current value of the register to be displayed. The ‘=’ function causes the number register to be set to the value specified, or to 0 if left null. The ‘+’ and ‘-’ functions cause the register to be incremented/decremented by ‘value’, or by 1 if value is null.

(.,.)p
The print command prints the addressed lines. '.' is left at the last line printed. The p command may be placed on the same line after any other command to cause printing of the last line affected by the command.

q
The quit command causes ed to exit. No automatic write of the file is done. If changes have been made to the current file since the last write command, you will be asked to repeat the quit command.

(.)r filename
The read command reads in the given file after the addressed line. If no file name is given, the remembered file name is used (see e and f commands). The remembered file name is not changed. Address '0' is legal

for this command and causes the file to be read in at the beginning of the buffer. If the read is successful, the number of lines read is typed. '.' is left at the last line read in from the file.

(.,.)s/regular expression/replacement/   or,
(.,.)s/regular expression/replacement/g

The substitute command searches each addressed line for an occurrence of the specified regular expression. On each line in which a match is found, the first occurrence of the expression is replaced by the replacement specified. If the global replacement indicator g appears after the command, all occurrences of the regular expression are replaced. Any character other than space or newline may be used instead of the slash '/' to delimit the regular expression and replacement. A question mark '?' is printed if the substitution fails on all addressed lines. '.' is left at the last line substituted.

An ampersand '&' appearing in the replacement is replaced by the string matching the regular expression. (The special meaning of '&' in this context may be suppressed by preceding it by '@'.)

The strings '$n', '$n+[d]' and '$n-[d]' appearing in the replacement string cause the current value of the number register to be placed in the line. The optional trailing increment/decrement syntax cause the number register value to incremented/decremented by `d' AFTER the value is placed in the string. If `d' is omitted, a value of 1 is used.

Lines may be split or merged by using the symbol '@n' to stand for the newline character at the end of a line.

t [format arguments]

This command allows one to `typeset' the current buffer without leaving the editor. The current contents of the buffer are written to a scratch file, and `format' is invoked with a command line consisting of the scratch file name plus any trailing arguments in the `t' command line. For example:

```
t +5 -7
```

causes format to be invoked on the buffer and pages 5 through 7 to be output. The value of '.' is not changed and the buffer is left intact.

(.)u

This causes the last line or range of lines which were deleted, either via a delete command or a substitute command, to be undeleted after the specified line. This is NOT an undo command. The last line or set of lines deleted are kept in a special place before recycling the line pointers, and may be recalled.

(1,$)w [>[>]]filename

The write command writes the addressed lines onto the given file. If the file does not exist, it is created. The remembered file name is not changed. If no file name is given, the remembered file name is used (see the e and f commands). '.' is left unchanged. If the command is successful, the number of lines written is typed. The form '>file' is equivalent to 'file', while '>>file' causes the lines to be appended to 'file'.

(1,$)x/regular expression/command

The except command is the same as the global command except that the command is executed for every line except those matching the regular expression.

(.)=

The line number of the addressed line is typed. '.' is left unchanged.

# comment

The remainder of the line after the "#" is a comment and ignored by the editor. This allows ed scripts to be commented for future enlightenment.

^shell command

The remainder of the line after the "^" is sent to the shell as a command. If there is nothing else on the line but a bare "^", the shell will be spawned, allowing a number of commands to be performed; when that shell quits, the terminal is returned to the editor. "." is left unchanged.

(.+1)<carriage return>

An address alone on a line causes the addressed line to be printed. A blank line alone is equivalent to '.+1' and thus is useful for stepping through text. A minus '-' followed by a carriage return is equivalent to '.-1'.

<file[-v]

The current input is stacked, 'file' is opened at READ access, and commands are read from ‘file’ until an EOF is encountered. If the optional −v flag is specified, each command is echoed on error output as it is executed. The normal search path is used to locate ‘file’, and a suffix of ".ed" is assumed. This facility is especially useful for canned procedures to be executed.

(1,$)|shell command
The remainder of the line after the "|" is spawned, with the lines specified fed to the command as its standard input. When the command completes, the terminal is returned to the editor. "." is left unchanged.

%
The percent of linepointers used is displayed. For in-memory versions of the editor, the percent of the in-memory character storage is also displayed.

SUMMARY OF SPECIAL CHARACTERS

The following are special characters used by the editor:

Character	Usage
?	Matches any character (except newline)
%	Indicates beginning of line
$	Indicates end of line or end of file
[...]	Character class (any one of these characters)
[!...]	Negated character class (any character except these characters)
{expression}	tagged pattern
*	Closure (zero or more occurrences of previous pattern)
+	Anchored closure (one or more occurrences)
@	Escaped character (e.g. @%, @[, @*)
&	Ditto, i.e. whatever was matched
c1−c2	Range of characters between c1 and c2

@f	Formfeed character
@l	Linefeed character
@n	Specifies the newline character at the end of a line
@r	Carriage return character
@t	Specifies a tab character

FILES

A temporary file is used to hold the text being edited. Two other temporary files, known as $1 and $2, may be used as parameters for the r, w, and @ commands. For example, if the current date and time are desired at the top of the text buffer, perform the following:

```
     * ^date >$1
     * 0r $1
```

As another example, if you wish to make a copy of lines 1,5 after the last line in the buffer, do the following:

```
     * 1,5w $1
     * $r $1
```

SEE ALSO

The Unix command "ed" in the Unix manual
The software tools tutorial "Edit"
"Edit is for Beginners" by David A. Mosher (available from
     UC Berkeley Computer Science Library)
"Edit: A Tutorial" (also available from the
     UC Berkeley Computer Science Library)
"A Tutorial Introduction to the ED Text Editor" by B. W. Kernighan
     (UC Berkeley Computer Science Library)
Kernighan and Plauger's "Software Tools", pages 163-217

DESCRIPTION

The error message "?" is printed whenever an edit command fails or is not understood.

AUTHORS

Original code by Kernighan and Plauger with modifications by Debbie Scherrer, Dennis Hall, Joe Sventek and Dave Martin.

BUGS/DEFICIENCIES

At the present time the editor is still in a somewhat experimental There is a compiled-in limit to the maximum number of lines which a file being edited may contain. The line limit applies to all lines read in and subsequently changed. This problem can be partly alleviated by writing (w command) and re-editing (e command) the file after a lot of lines have been changed.

There are several discrepancies between this editor and Unix's ed. These include:

1. Unix uses 'v' instead of 'x' for the except command.

2. Unix uses '^' instead of '%' for the beginning-of-line character.

3. Unix uses '.' instead of '?' to indicate a match of any character.

4. Unix uses '^' instead of '!' to indicate exclusion of a character class.

5. Unix uses '\' instead of '@' for the escape character.

6. Unix uses '?' instead of '\' to delimit a backward search pattern.

7. The Unix 'r' command uses the last line of the file, instead of the current line, as the default address.

8. The Unix editor prints the number of characters, rather than lines read or written when dealing with files.

NAME

Entab - convert spaces to tabs and spaces

SYNOPSIS

```
entab [<t1>...] [+<n>] [file] ...
```

DESCRIPTION

Entab replaces strings of blanks with equivalent tabs (control-i) and blanks. It can be used to read files and produce typewriter-like text, reducing file size. Tab stops are indicated by <t1> ... (default 8, 16, ...), while +<n> indicates tab stops every <n> columns. Thus the command

```
entab 5 21 +5
```

would insert tab stops at columns 5, 21, 26, etc. If no files are specified, the standard input is read. An isolated minus sign also indicates the standard input.

SEE ALSO

detab
lpr

AUTHORS

Original from Kernighan & Plauger's 'Software Tools', with modifications by Dennis Hall.

BUGS/DEFICIENCIES

NAME

Esh - extended shell, with intraline editing and history

SYNOPSIS

esh [-cdnvx] [file [arguments]]

DESCRIPTION

'esh' is an extended version of 'sh' which incorporates several features designed to make it easier to use.

L I N E   E D I T I N G

- Both backspace (^H) and RUBOUT (RUB, DEL) may be used to delete the last character typed.

- ^U may be used to undo the current line - i.e. delete it and re-prompt for the line.

- ^R may be used to re-type the line. This is useful when working on a hard-copy terminal, since character deletes are done with backspaces.

- ^W deletes the last word, where words are defined as strings of non-blanks.

- ^D causes the current working directory to be listed on the terminal, after which the line is re-displayed and you may continue input on the current line. This is useful when you get part way through a command, and then realize that the critical file name has slipped from recent memory.

- ^F (or ESC) causes file recognition to be performed on the current pathname. If the filename can be extended unambiguously, it will be; otherwise, a list of files matching the current pattern are displayed, the line re-displayed, and you may continue input on the line.

- ^A causes the previous command line to be retrieved and the cursor to be positioned at the end. This is useful for adding stages to pipelines, for example. ^A may also be used in conjunction with the history mechanism to append to previous commands.

- ^E causes the intraline editor to be entered. If the cursor is at the beginning of a line the previous line is retrieved; otherwise the current line is edited. The editing commands are discussed below in the section on intraline editing.

H I S T O R Y   M E C H A N I S M

A history of the commands input to 'esh' are maintained for each session. You may invoke special history manipulating functions by starting a command line with an exclamation mark (! - also known as a BANG) in column 1. If is is necessary to send a line starting with a BANG to the shell, lines starting with "@!" have the "@" stripped off, and the remainder of the line is given to the shell.

Lines starting with BANG enable you to communicate with a miniature version of the editor 'ed'. At any time, the last 25 commands are available for recall and manipulation. The current line concept of 'ed' is supported, although the current line is ALWAYS the last command in the history. Legal history commands are:

1. history display

   !h[istory] [n][l]

   This is the equivalent of a browse command in 'ed'. !h will display the last screenful of commands, along with their line numbers. The screensize, which defaults to 22 lines, may be changed by specifying a BLANK and a number following the !h[istory] string (!h 10, for example). The new screensize is remembered and used in all !h commands as the default screensize. Specifying a screensize larger than 25 has the effect of setting the size to 25. The optional trailing 'l' (list) will cause control characters in the commands to be displayed as '^<char>', where <char> is the character one needs to type in conjunction with the CTRL key to generate the control character.

   !b[rowse] [n][l]

   This command is a synonym for history. It is included to increase the similiarity of function with the editor.

2. history recall

   ![line_number][;line_number]...

   This command permits the recall of a command from the history for re-execution. The command so recalled is displayed and then passed on to the shell for execution. This command is then entered at the bottom of the history.

Valid line_numbers are the same as those for the editor. For example, a line_number may be the number listed next to the command in the history display, a pattern of the form "\pattern[\]", which indicates a backward search in the 25 line history window, or a pattern of the form "/pattern[/]", indicating a search forward, wrapping to the start of the 25 line window. The trailing '\' or '/' are optional when specifying a single pattern. The semi-colon syntax is the same as that in 'ed', indicating that the search for the second pattern is to start at the line where the first pattern was found.

If the pattern specified was illegal, or a line matching the pattern could not be found, or an invalid line_number was specified, a comment is displayed

```
# invalid line number
```

and you are prompted for more input. The history is not modified in this case.

All sequences of patterns resolve into a single line number. It is not possible to request a range of lines from the history.

It should be noted that the line_numbering is completely regular with 'ed'. In particular, "!" followed by nothing maps into a fetch of the current line (last command typed). See the writeup on 'ed' for more details on the specification of line_numbers.

3. history recall and modification

```
![line_number]s/pat/repl[/[g]]
```

Upon successfully recalling a command from the history, it may be modified before it is passed on to 'esh' for execution. This is performed with the 's' command, which is exactly the same as that for 'ed'. The delimiters for 'pat' and 'repl' may be any character, the remembered pattern feature is available, and the trailing delimiter after the replacement pattern is optional. The optional trailing 'g' indicates substitution for all occurrences of 'pat' in the line. See the 'ed' manual entry for more information on the substitute command.

If the substitution fails for any reason, a comment is displayed

```
# illegal substitution
```

and you are prompted for more input. The history is not modified in this case.

4. history archiving

   ```
   !w[rite] [>[>]]file
   ```

   This command permits you to archive (save) the entire transcript of activity to a file. It also passes an EOF to 'esh', which causes 'esh' to terminate execution. The commands

   ```
   !w file
   !w >file
   ```

   both cause 'file' to be overwritten with the transcript, while >>file causes the transcript to be appended to 'file'.

   It should be noted that the !w command causes ALL of the input given to 'esh' in this session to be saved, not just the current 25 line window. It also passes an EOF to 'esh', which will terminate execution.

5. history deletion

   ```
   !q[uit]
   ^Z
   ```

   These commands cause an EOF to be sent to 'esh' and the deletion of the log of activity.

Lines consisting solely of a carriage return are NOT logged in the history. If you need to perform several edits on a command before having it executed, you can exploit the fact that lines beginning with a sharp (#) are comments to the shell. For example:

```
!\%ed\s/%/#/                            <make it a comment>
!s/pat1/repl1/                          <still a comment  >
      .                                      .
      .                                      .
      .                                      .
!s/patn/repln/                          <still a comment  >
!s/%#//                                 <now execute it   >
```

All of the intermediate comment lines will be placed in the history, displacing other lines from the window which may possibly be needed. Of course, it may be simpler in such cases to just enter the command by hand.

I N T R A L I N E   E D I T I N G

The intraline editing functions are a subset of those available in the "VI" screen editor from Berkeley. You are referred to the VI documentation for a tutorial introduction.

The intraline editing "mode" is entered via ^E. Exactly what happens when the ^E is typed depends on what precedes it on the command line. If the ^E is the first character on a line, the previous command is retrieved and the cursor is positioned at the beginning of the line. If the line is a history reference (i.e. begins with a "!"), the referenced line is retrieved and the cursor is positioned at the beginning of the line. If the line is anything else, the cursor is positioned at the end of the line.

Once in the intraline editor the following commands are allowed:

```
Notes:  '[n]' indicates an optional integer count
        <text> input is terminated with ^Z or ESC
```

```
MOVE cursor:
------------

[n]SPACE        -> <n> positions
[n]BS           <- <n> positions
[n]h            <- <n> positions
%          <- to beginning of line (BOL)
$          -> to end of line (EOL)
[n]w            -> <n> (non-alphanumeric) words
[n]W            -> <n> (non-blank)        words
[n]b            <- <n> (non-alphanumeric) words
[n]B            <- <n> (non-blank)        words
[n]e            -> to end of <n>th (non-alphanumeric) word
[n]E            -> to end of <n>th (non-blank)        word

[n]f<c>         -> thru <n>th occurrence of char <c>
[n]t<c>         -> to   <n>th occurrence of char <c>
[n]F<c>         <- thru <n>th occurrence of char <c>
[n]T<c>         <- to   <n>th occurrence of char <c>
[n];            Repeat last 'f', 't', 'F', or 'T'
[n],            Repeat last 'f', 't', 'F', or 'T' in reverse
```

```
     INSERT or APPEND <text>:
     ------------------------

     [n]i<text>      Insert text before cursor
     [n]I<text>      Insert text before beginning of line
     [n]a<text>      Append text after  cursor
     [n]A<text>      Append text after  end of line


     REPLACE or SUBSTITUTE <text> for character(s):
     ----------------------------------------------

R<text>         Replace (overlay) text on screen with <text>
r<c>            Replace current  character with <c>

     [n]s<text>      Substitute <n> characters with <text>


     CHANGE <text object> to <text>:
     -------------------

     [n]cw<text>     next <n> (non-alphanumeric) words to <text>
     [n]cW<text>     next <n> (non-blank)        words to <text>
     [n]ce<text>     thru end of <n>th (non-alphanumeric) word to <text>
     [n]cE<text>     thru end of <n>th (non-blank)        word to <text>
  c%<text>      text from BOL thru cursor to <text>
  c$<text>      text from cursor thru EOL to <text>
  C<text>       Synonym for 'c$'


     DELETE <text object>(s):
     ------------------------

     [n]x            <n> characters, starting at cursor
     [n]dSPACE       <n> characters, starting at cursor
     [n]X            previous <n> characters
     [n]dw           next     <n> (non-alphanumeric) words
     [n]dW           next     <n> (non-blank)        words
     [n]db           previous <n> (non-alphanumeric) words
     [n]dB           previous <n> (non-blank)        words
     [n]df<c>        thru next <n>th occurrence of char <c>
     [n]dt<c>        to   next <n>th occurrence of char <c>
     [n]dF<c>        thru prev <n>th occurrence of char <c>
     [n]dT<c>        to   prev <n>th occurrence of char <c>
  dd            entire line
  d%            from beginning of line to cursor, inclusive
  d$            from cursor to end of line, inclusive
  D             Synonym for 'd$'
```

[n].	Repeat previous ‘delete’ command

UNDO action of previous command(s):
----

u	Undo the last change to the line
U	Undo ALL commands; restore line to original state

EXIT intra-line editor:
----

^Z	Move cursor to EOL and exit intra-line edit
^E	Move cursor to EOL and force RETURN
RETURN	Delete after cursor to EOL and execute command line

The three methods of exiting the intraline editing mode are worthy of special mention. In particular you will usually exit with ^E rather than RETURN or ^Z, since the RETURN will chop off everything to the right of the cursor and ^Z will merely return to the line-gathering routine which invoked the intraline editor. Note that a ^E^E sequence may be used to repeat the previous command line.

FILES

SEE ALSO
sh - command line interpreter

DIAGNOSTICS
# invalid line number
# invalid substitution

AUTHORS
Editing features: Dave Martin
History mechanism: Joe Sventek

BUGS/DEFICIENCIES

NAME

Exist - check for the existence of a file

SYNOPSIS

exist file

DESCRIPTION

exist attempts to open the named file at READ access. If successful, it closes the file and returns the value of 1 in the DCL symbol $STATUS. Common uses are for system-wide login files for the invocation of your login.com file as in the following:

```
$ exist:==$st_bin:exist
$ exist login.com
$ if $STATUS.eq.1 then @login
```

FILES

SEE ALSO

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME
    Expand - uncompress input files

SYNOPSIS
    expand [file] ...

DESCRIPTION
    Expand expands files previously compressed by 'cpress'. If no input files are given, or if the filename '-' appears, input will be read from the standard input.

FILES

SEE ALSO
    cpress

DIAGNOSTICS
    A message is printed if an input file cannot be opened; further processing is terminated.

AUTHORS
    Original from Kernighan & Plauger's 'Software Tools', with minor modifications by Debbie Scherrer.

BUGS/DEFICIENCIES

NAME

Fb - search blocks of lines for text patterns

SYNOPSIS

```
fb [-acix] [-ln] [-sexpr [-sexpr]] expr [expr] ...
```

DESCRIPTION

"Fb" (find block) searches blocks or groups of lines in a file for text patterns. It is similar to 'find' except that if a pattern is found, the entire block of lines is copied to standard output, rather than simply the line in which the pattern occurred. Thus it is useful for searching mailing lists, bibliographies, and similar files where several lines are grouped together to form cohesive units.

The search patterns may be any regular expression as described in the 'ed' and 'find' writeups.

"Fb" assumes the blocks of lines are separated by an empty line or a line containing only blanks. When "fb" is called without any options, standard input is read and each line is checked to see if it matches any of the regular expressions given as arguments. If any matches are found, the entire block is printed on standard output.

Other options include:

-a	Only print the block if ALL the arguments are found within it
-x	Only print the block if none of the arguments are found within it
-c	Only print a COUNT of the number of blocks found which match/don't match the expressions
-i	Perform the pattern matches ignoring case.
-sexpr	Use 'expr' as the block separator (instead of a blank or empty line). "Expr" can be a regular expression just as the search arguments can. If two "-sexpr" arguments are given, the first one is considered to be the pattern which starts a block (e.g. -ssubroutine) and the second is considered the pattern which ends a block (e.g. -send). If the -i flag has been seen before the -s flags, then the start and end expressions will be case-independent.
-ln	prints only the first 'n' lines of the block; if the

block contains less than 'n' lines, the block is padded out with blank lines.

Care should be taken when using the characters % $ [ ] ! * @ and any shell characters in the text pattern. It is often necessary to enclose the entire substitution pattern in quotes.

FILES

A scratch file ("fbt") is used if the internal line buffer becomes full.

SEE ALSO

find
ed

For a complete description of regular expressions, see "Software Tools" pages 135-154.

DIAGNOSTICS

Error messages are given if:

a) One of the patterns given is illegal
b) Too many separators are given (2 are allowed)
c) The maximum number of expressions is exceeded (9 are allowed)
d) There are problems opening the scratch file (when the block line buffer fills up).

If the following messages show up, something is dreadfully wrong:

a) "Illegal default separator"
b) "Block buffer overflow"

AUTHORS

Debbie Scherrer (Lawrence Berkeley Laboratory)

BUGS/DEFICIENCIES

An expression may not start with a minus sign (-).

Regular expressions cannot span line boundaries.

NAME

Fc - fortran compiler

SYNOPSIS

fc [-cdmov] file ...

DESCRIPTION

fc is the fortran compiler callable from the software tools shell. It accepts the following types of arguments:

1. Files whose names end in '.f' are assumed to be fortran source programs. They are compiled, and the object file is left on a file whose name is that of the source with '.obj' substituted for '.f'.

2. Other arguments (except for the flags listed in 3 below) are assumed to be either loader flags, or object files, typically created by an earlier fc run. These programs, together with the results of any compilations, are loaded (in the order given) to produce an executable program.

3. The flags which affect the actions of the compiler are:

   -c suppress the loading phase, as does any compilation error in any routine

   -d do whatever is necessary to prepare the object files for the system-specific debugger. This flag is passed on to 'ld' if the -c switch is not specified.

   -m passed on to 'ld' to cause a load map to be produced.

   -o generates a fortran listing for 'file.f' on 'file.l'

   -v verbose option; prints additional information about the compilation process

SEE ALSO

rc, the ratfor compiler, which provides a more pleasant programming dialect and environment

ld, the loader, for descriptions of loader flags and process naming conventions

AUTHORS

Joe Sventek wrote the interface of fc to the DEC ForTran compiler.

BUGS/DEFICIENCIES

NAME

Fd - fast directory list in sort order

SYNOPSIS

```
fd [path] ...
```

DESCRIPTION

'fd' lists the files matching the specified pattern in sort order, packed in 5 columns across the page. The packing occurs regardless of whether standard output is a terminal or not, in contrast to the actions of 'ls'. If no 'path' arguments are specified, all files in the current working directory are listed. The forms of 'path' are identical to those for 'ls'.

FILES

SEE ALSO

ls - general directory listing tool

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME

Field - manipulate fields of data

SYNOPSIS

```
field [-t[c] | fieldlist] outputformat [file] ...
```

DESCRIPTION

field is used to manipulate data kept in formatted fields. It selects data from certain fields of the input files and copies it to certain places in the standard output.

The 'fieldlist' parameter is used to describe the interesting columns on the input file. Fields are specified by naming the columns in which they occur (e.g. 5-10) or the columns in which they start and an indication of their length (e.g. 3+2, meaning a field which starts in column 3 and spans 2 columns). When specifying more than one field, separate the specs with commas (e.g. 5-10,16,72+8) Fields may overlap, and need not be in ascending numerical order (e.g. 1-25,10,3 is OK).

If input fields do not fall in certain columns, but rather are separated by some character (such as a blank or a comma), describe the fields by using the '-tc' flag, replacing 'c' with the appropriate separator (a tab character is the default).

Once fields have been described with either the '-tc' flag or a fieldlist, they can be arranged on output by the 'outputformat' argument. This argument is actually a picture of what the output line should look like. Fields from input are referred to as $1, $2, $3, etc., referring to the first, second, third, etc. fields that were specified. (Up to 9 fields are allowed, plus the argument $0 which refers to the whole line.) These $n symbols are placed in the output format wherever that field should appear, surrounded by whatever characters desired. For example, an outputformat of:

```
                  "$2 somewords $1"
```

would produce an output line such as:

```
              field2 somewords field1
```

If no input files are specified, or if the filename '-' is found, field will read from the standard input.

DIAGNOSTICS

illegal field specification

The fieldlist specification was in error, probably because it contained letters or some other illegal characters

SEE ALSO

sedit

AUTHORS

David Hanson and friends (U. of Arizona)

NAME

Find - search a file for text patterns

SYNOPSIS

```
find [-acix] expression [expression] ...
```

DESCRIPTION

find searches the standard input file for lines matching the text patterns "expression" (up to 9 patterns may be specified) according to the matching criterion specified by the switches. (A text pattern is a subset of a "regular expression"--see the writeup on "ed" for a complete description of regular expressions.) Unless the -c option is specified, each matching line is copied to the standard output.

By default, any line which matches any one of the expressions is considered a matching line. If the -a flag is specified, only lines which match all expressions in any order are considered to match. If the -x flag is specified, all lines which don't satisfy the above criteria are considered matching lines. If the -c option is specified, matching lines are counted instead of being copied to the standard output, and the final count is written to the standard output. Finally, if the -i option is specified, the pattern matching becomes case insensitive.

A text pattern consists of the following elements:

```
c      literal character
?      any character except newline
%      beginning of line
$      end of line (null string before newline)
[...]  character class (any one of these characters)
[!...] negated character class (all but these characters)
*      closure (zero or more occurrences of previous pattern)
+      anchored closure (one or more occurrences of previous pattern)
@c     escaped character (e.g., @%, @[, @*)
```

Any special meaning of characters in a text pattern is lost when escaped, inside [...], or for:

```
%         not at beginning
$         not at end
*         at beginning
+         at beginning
```

A character class consists of zero or more of the following elements, surrounded by [and]:

```
c         literal character, including [
```

a-b	range of characters (digits, lower or upper case)
!	negated character class if at beginning
@c	escaped character (@! @- @ @])

Special meaning of characters in a character class is lost when escaped or for

!	not at beginning
-	at beginning or end

An escape sequence consists of the character @ followed by a single character:

@f	formfeed
@l	linefeed
@n	newline
@r	carriage return
@t	tab
@c	c (including @)

For a complete description, see "Software Tools" pages 135-154. Care should be taken when using the characters % $ [ ] ! * + @ and any shell characters in the text pattern. It is often necessary to enclose the entire substitution pattern in quotes.

FILES

none

SEE ALSO

tr, ed, ch and the UNIX grep command.
xfind - extended find utility

DIAGNOSTICS

An error message is printed if one of the patterns given is illegal.

AUTHORS

Originally from Kernighan & Plauger's "Software Tools", with major modifications by Joe Sventek.

BUGS/DEFICIENCIES

An expression may not start with a minus sign(-).

NAME

Form - produce form letter by prompting user for information

SYNOPSIS

```
form [-c] [+c] file ...
```

DESCRIPTION

Form reads input files and writes them to the standard output. Any time it encounters some characters surrounded by angle brackets ('<' and '>') it prints the string between the characters as a prompt to the user. It then reads from the standard input and replaces the bracketed string with what was read.

Normally only one line of input is accepted from the standard input. However, a response can be continued on succeeding lines by terminating each line to be continued with a minus ('-').

Multi-line input will also be accepted if the left-most bracketing character of the prompt is immediately followed by a minus ('-') as in <-long prompt>. Upon detection of a prompt of this form, the input can only be terminated by typing a bare period ('.') on a line, as in the editor. This fact is brought to the user's attention when the prompt is displayed.

The prompts inside the file may also span line boundaries if so desired.

The user's answers to prompts are remembered, so duplicate prompts are replaced without repeating the prompt to the user.

If the standard input is not a terminal, no prompts are issued.

The '-c' flag may be used to reset the initial character signalling a prompt. The character 'c' then replaces the '<'.

The '+c' flag may be used to reset the terminating character of a prompt. The character 'c' then replaces '>'.

It is possible to have 'form' ask for and fill in repeated fields in your document. If a prompt of the form

```
<REPEAT label>
```

is detected, all of the lines up to one consisting of <label> will be repeated. The number of repetitions is requested from the user by the prompt

Count for REPEAT label

REPEAT loops may be nested up to a depth of five (5). If if is necessary to specify a prompt <REPEAT> which does not have the special meaning of starting a loop, <\REPEAT> will have the leading '\' stripped off, and the prompt will be used normally.

It is necessary to place the <REPEAT label> starting directive and the <label> terminating directive on lines by themselves.

FILES
Your terminal is opened at READ access.

SEE ALSO
The Unix form-letter tool

DIAGNOSTICS
If an input file cannot be opened, a message is printed and execution is terminated.

A message is also printed if either the prompt or the response is too long for the tool's internal buffer.

AUTHORS
Debbie Scherrer

BUGS/DEFICIENCIES

NAME
Format - format (roff) text

SYNOPSIS
```
format [+n] [-n] [-s] [-pon] [file] ...
```

DESCRIPTION
Format formats text according to request lines embedded in the text of the given files or standard input if no files are given. If nonexistent filenames are encountered they are ignored. The optional flags are as follows:

+n Start printing at the first page with number "n".

-n Stop printing at the first page numbered higher than "n".

-s Stop before each page, including the first (useful for paper manipulation). The prompt "Type return to begin a page" is given just once before the first page. For each page thereafter, the terminal bell is rung to indicate that another sheet of paper is needed.

-pon Move the entire document "n" spaces (default=0) to the right ("page offset").

Input consists of intermixed text lines, which contain information to be formatted, and request lines, which contain instructions about how to format the text lines. Request lines begin with a distinguishing "control character", normally a period.

Output lines are automatically "filled"; that is, their right margins are justified, without regard to the format of the input text lines. (Right justification may be turned on and off through the use of the ".ju" and ".nj" commands, though.) Strings of embedded spaces are retained so that the output line will contain at least as many spaces between words as the input line. However, input lines beginning with a space are output without modification.

Line "breaks" may be caused at specified places by certain commands, or by the appearance of an empty input line or an input line beginning with a space.

Because of the nature of its output (backspace and tab characters and a fixed number of lines per page), it is generally necessary to have a tool developed especially for printing the output on the local printers. On most systems this is a combination of the tools 'os' and 'detab', plus some sort of page eject control of the printer. If such as tool exists, it should be described in Section 3 of this manual.

The capabilities of format are specified in the attached Request Summary. Numerical values are denoted by "n", titles by "t", and single characters by "c". Numbers may be signed + or -, in which case they signify relative changes to a quantity; otherwise they signify an absolute setting. Missing "n" fields are ordinarily taken to be 1, missing "t" fields to be empty, and "c" fields to shut off the appropriate special interpretation.

Running titles may appear at the top and bottom of every page. A title line consists of a line with three distinct fields: the first is text to be placed flush with the left margin, the second centered, and the third flush with the right margin. The first non-blank character in the title will be used as the delimiter to separate the three fields. Any "#" characters in a title are replaced by the current page number, and any "%" characters are replaced by the current date.

The ".nr" defines number registers; there are 26 registers named a-z. The command ".nr x m" sets number register x to m; ".nr x +m" increments number register by m; and ".nr x -m" decrements x by m. The value of number register x is placed in the text by the appearance of @nx; a literal @ may be inserted using @@.

Additional commands may be defined using ".de xx". For example,

```
.de PG
.sp
.ti +3
.en
```

defines a "paragraph" command PG. Defined commands may also be invoked with arguments. Arguments are separated by blanks or tabs. Within the definition of a defined command, arguments are referenced using $1, $2, etc. There is a maximum of 9 arguments. Omitted arguments default to the null string. $0 references the command name itself. For example, the following version of the paragraph command uses the argument to determine the amount of indentation.

```
.de PG
.sp
.ti +$1
.en
```

This command could be invoked by

```
   .PG 3
```

to get the same effect as the previous version.

The ".so file" command causes the contents of file to be inserted in place of the ".so" command; ".so" commands may be nested.

FILES

none

SEE ALSO

Kernighan & Plauger's "Software Tools", pages 219-250
whatever tool has been devised for printing formatted output
The roff and nroff/troff UNIX commands
The "nroff" and "troff" users manuals by Joseph F. Ossana, Bell Laboratories, Murray Hill, New Jersey

DIAGNOSTICS

invalid number register name
   names of number registers must be a single letter a-z

missing name in command definition
   a macro was defined using the '.de' command, but no 2-letter name for it was given

so commands nested too deeply
   the limit for nesting included source files is dependent upon the MAXOFILES definition in the standard symbols definition file

too many characters pushed back
   the buffer holding input characters has been exceeded; its size is determined by the BUFSIZE definition in the source code

AUTHORS

Original version by Kernighan and Plauger, with modifications by David Hanson and friends (U. of Arizona), Joe Sventek and Debbie Scherrer (Lawrence Berkeley Laboratory)

REQUEST SUMMARY

Request	Initial	Default	Break	Meaning
.##				start of comment line
.bd n		n=1	no	boldface the next n lines
.bp n	n=1	n=+1	yes	begin new page and number it n
.br			yes	break
.cc c	c=.	c=.	no	control character becomes c
.ce n		n=1	yes	center the next n input lines
.cu n		n=1	no	continuously underline in the next n
.de xx			no	command xx; ends at .en
.ef t	t=""	t=""	no	foots on even pages are t
.eh t	t=""	t=""	no	heads on even pages are t
.en			no	terminate command definition
.fi	yes		yes	begin filling output lines
.fo /l/c/r	f=""	f=""	no	foot titles are l(eft), c(enter), r(ight)
.he /l/c/r	t=""	t=""	no	head titles are l(eft), c(enter), r(ight)
.in n	n=0	n=0	yes	set left margin to column n+1
.ju	yes	yes	no	begin justifying filled lines
.ls n	n=1	n=1	no	set line spacing to n
.m1 n	n=3	n=3	no	space between top of page and head
.m2 n	n=2	n=2	no	space between head and text
.m3 n	n=2	n=2	no	space between text and foot
.m4 n	n=3	n=3	no	space between foot and bottom
.ne n		n=0	y/n	need n lines; break if new page
.nf	no		yes	stop filling
.nj	no		no	stop justifying
.nr x m	x=0	m=0	no	set number register x to m, -m, +m for decrement, increment
.of t	t=""	t=""	no	foots on odd pages are t
.oh t	t=""	t=""	no	heads on odd pages are t
.pl n	n=66	n=66	no	set page length to n lines
.po n	n=0	n=0	no	set page offset to n spaces
.rm n	n=65	n=65	no	set right margin to column n
.so file			no	switch input to file
.sp n		n=1	yes	space n lines, except at top of page
.st n		n=0	yes	space to line n from top; -n spaces to line n from bottom
.ti n		n=0	yes	temporarily indent next output line n spaces
.ul n		n=1	no	underline words in the next n input lines

NAME

Get - get generation from TCS file

SYNOPSIS

get [-h][-rM.N] historyfile

DESCRIPTION

Get retrieves earlier versions of text from "historyfile" as computed by DELTA.

The possible flags are:

(none) - The latest version of the file is retrieved.

-h - Print out the history information associated with the versions. The dates, times, and user IDs will be retrieved, along with the comments added while performing the DELTAs.

-rM.N - Retrieve the specified version M.N.

The retrieved version of the file will be sent to the standard output. History information is always sent to the terminal.

FILES

SEE ALSO

admin, delta

DIAGNOSTICS

usage: get [-h][-rM.N] historyfile
Correct calling format is provided when called without arguments.

Unexpected EOF on history-info scan.
The source file does not contain the code which identifies it as a TCS history file. The code may be entered via the ADMIN command.

Unexpected EOF on history-data scan.
The file format has been tampered with and is no longer recognizable. Refer to a guru for repair.

- missing from keyletter
First argument is expected to qualify whether versions and/or histories are to be extracted.

Illegal keyletter
Only 'h' and 'r' are valid keys.

Nonexistant revision level requested.
: The version number specified is not contained in the history. Try "get -h file.tcs" to view the versions available.

Invalid history file
: The history file specifies impossible line-number correlations. Either out-of-sequence changes or line numbers in descending order.

Cannot locate TCS history file.
: Could not find file supplied for historyfile.

AUTHORS

An Algorithm for Differential File Comparison by J.W.Hunt and M.D.McIlroy (BTL Computing Science Technical Report #41). Original code by Wil Baden; converted from MORTRAN by Dave Murray. Modifications and conversion to BTL-SCCS style by Neil Groundwater at ADI. The Source Code Control System was introduced by Marc J. Rochkind in the December, 1975, IEEE Transactions on Software Engineering.

BUGS/DEFICIENCIES

NAME

Grep - search file[s] for a pattern

SYNOPSIS

```
grep [-chilx] expression [file] ...
```

DESCRIPTION

'grep' searches the names files (or standard input if none are specified) for occurrences of the expression. The set of valid expressions are the same as those for 'find', 'ch' and 'ed'. The manual entries for those tools may be consulted for full details. The output of 'grep' is dependent upon which switches are selected:

None	When one or more occurrences of the expression are found in a file, the file name is displayed, with each line in which the expression occurs listed below the file name.
-c	Only the number of matching lines in each file is displayed.
-h	Do not display the file names.
-i	Make comparisons case insensitive.
-l	Only the names of files which contain matching lines are displayed, one per line.
-x	Display (or count) only those lines which do NOT contain the expression.

FILES

none

SEE ALSO

find - find groups of expressions in a file
ch - globally change expressions within a file
ed - text editor

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME

Hsh - shell with history and editing functions

SYNOPSIS

hsh [-cdnvx] [file [arguments]]

DESCRIPTION

'hsh' is identical to 'sh' with the exception that a history is kept of commands typed; recall and editing functions on the history are permitted and described below. Consult the manual entry for 'sh' for more information on the common functions.

A history of the commands input to 'hsh' are maintained for each session. The user may invoke special history manipulating functions by starting a command line with an exclamation mark (! - also known as a BANG) in column 1. If is is necessary to send a line starting with a BANG to the shell, lines starting with "@!" have the "@" stripped off, and the remainder of the line is given to the shell.

Lines starting with BANG enable the user to communicate with a miniature version of the editor 'ed'. At any time, the last 25 commands are available for recall and manipulation. The current line concept of 'ed' is supported, although the current line is ALWAYS the last command in the history. Legal history commands are:

1. history display

   !h[istory] [n][l]

   This is the equivalent of a browse command in 'ed'. !h will display the last screenful of commands, along with their line numbers. The screensize, which defaults to 22 lines, may be changed by specifying a BLANK and a number following the !h[istory] string (!h 10, for example). The new screensize is remembered and used in all !h commands as the default screensize. Specifying a screensize larger than 25 has the effect of setting the size to 25. The optional trailing 'l' (list) will cause control characters in the commands to be displayed as '^<char>', where <char> is the character one needs to type in conjunction with the CTRL key to generate the control character.

   !b[rowse] [n][l]

   This command is a synonym for history. It is included to increase the similiarity of function with the editor.

2. history recall

   ```
   ![line_number][;line_number]...
   ```

   This command permits the recall of a command from the history for re-execution. The command so recalled is displayed to the user and then passed on to the shell for execution. This command is then entered at the bottom of the history.

   Valid line_numbers are the same as those for the editor. For example, a line_number may be the number listed next to the command in the history display, a pattern of the form "\pattern[\]", which indicates a backward search in the 25 line history window, or a pattern of the form "/pattern[/]", indicating a search forward, wrapping to the start of the 25 line window. The trailing '\' or '/' are optional when specifying a single pattern. The semi-colon syntax is the same as that in 'ed', indicating that the search for the second pattern is to start at the line where the first pattern was found.

   If the pattern specified was illegal, or a line matching the pattern could not be found, or an invalid line_number was specified, a comment is displayed to the user

   ```
   # invalid line number
   ```

   and the user is prompted for more input. The history is not modified in this case.

   All sequences of patterns resolve into a single line number. It is not possible to request a range of lines from the history.

   It should be noted that the line_numbering is completely regular with 'ed'. In particular, "!" followed by nothing maps into a fetch of the current line (last command typed). See the writeup on 'ed' for more details on the specification of line_numbers.

3. history recall and modification

   ```
   ![line_number]s/pat/repl[/[g]]
   ```

   Upon successfully recalling a command from the history, it may be modified before it is passed on to 'hsh' for execution. This is performed with the 's' command, which is exactly the same as that for 'ed'. The delimiters for 'pat' and 'repl' may be any character, the remembered pattern feature is available, and the trailing delimiter after the replacement pattern is optional. The optional trailing 'g' indicates substitution for all occurrences of 'pat' in the line. See the 'ed' manual entry for more information on the substitute command.

If the substitution fails for any reason, a comment is displayed to the user

```
# illegal substitution
```

and the user is prompted for more input. The history is not modified in this case.

4. history archiving

```
!w[rite] [>[>]]file
```

This command permits the user to archive (save) the entire transcript of activity to a file. It also passes an EOF to 'hsh', which causes 'hsh' to terminate execution. The commands

```
!w file
!w >file
```

both cause 'file' to be overwritten with the transcript, while >>file causes the transcript to be appended to 'file'.

It should be noted that the !w command causes ALL of the input given to 'hsh' in this session to be saved, not just the current 25 line window. It also passes an EOF to 'hsh', which will terminate execution.

5. history deletion

```
!q[uit]
^Z
```

These commands cause an EOF to be sent to 'hsh' and the deletion of the log of activity.

Lines consisting solely of a carriage return are NOT logged in the history. If the user needs to perform several edits on a command before having it executed, he can exploit the fact that lines beginning with a sharp (#) are comments to the shell. For example:

```
     !\%ed\s/%/#/                      <make it a comment>
     !s/pat1/repl1/                    <still a comment  >
          .                                 .
          .                                 .
          .                                 .
     !s/patn/repln/                    <still a comment  >
     !s/%#//                           <now execute it   >
```

All of the intermediate comment lines will be placed in the history, displacing other lines from the window which may possibly be needed. Of course, it may be simpler in such cases to just enter the command by hand.

FILES

Creates a scratch file ~tmp/pid.log for the command transcript.

SEE ALSO

sh - command line interpreter
esh - shell with file recognition and RAW tty I/O
ed - text editor

DIAGNOSTICS

# invalid line number
# invalid substitution

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

Due to address space limitations on some systems (RSX-11M, for example), the history shell will not have support for the following shell internal commands:

alias ask param unalias unparam source

For those systems where the 'source' command is supported, commands read from alternate input files (login.sh, 'source'ed files) are NOT logged in the history.

NAME

Incl - expand included files

SYNOPSIS

incl [file] ...

DESCRIPTION

Include copies the input files to the standard output. Whenever an input line begins with

include filename

the entire contents of filename will be copied to the standard output. If no input files are specified, the standard input is copied. An included file may include further includes. Multiple input files are allowed. Include is used to bring in much-used routines, common declarations or definitions, thus insuring use of the same version by all programs.

FILES

none

SEE ALSO

Kernighan and Plauger's "Software Tools", pages 74-77.
The software tools "ratfor" tutorial

DIAGNOSTICS

includes nested too deeply

The depth of included files allowed is dependent upon the maximum number of open files allowed in the following manner:

MAXOFILES - 3

filename: can't open

File could not be located or maximum number of opened files was exceeded.

AUTHORS

Original code by Kernighan and Plauger in "Software Tools", with modifications by Ardith Kenney.

BUGS/DEFICIENCIES

The depth of included files allowed is dependent upon the maximum number of open files allowed by the implementor of the primitives.

NAME

Isam - generate index for pseudo-indexed-sequential access

SYNOPSIS

```
isam [-d<dif>] [-w<width>] [-j<l/r>]
```

DESCRIPTION

isam is used to generate an index for a text file such that the index may be used later to permit indexed-sequential access to the file. isam reads every 'dif'th line (default is 1) from the standard input, noting its disk address with a call to note. It uses getwrd to retrieve the first "word" from the line and uses this as the primary key to the record. This key is then output to standard output in a field 'width' wide (default is 25) and justified according to the -j switch (default left). The two-word address from note is then output as decimal integers before the index record is flushed.

FILES

SEE ALSO

spell - spelling error finder; uses an isam-generated index
asam - generate index for archives

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME
    Kill - kill a running process

SYNOPSIS
    kill processid [processid ...]

DESCRIPTION
    kill kills the processes specified by the processid's in the command line. The processid's are those provided by the shell when it spawns a background process.

FILES
    none

SEE ALSO
    sh - shell (command line interpreter)

DIAGNOSTICS
    if the process specified by the processid does not exist, an error message will be displayed on error output.

AUTHORS
    Joe Sventek (VAX)

BUGS/DEFICIENCIES

NAME

Kwic - make keyword in context index

SYNOPSIS

kwic [file] ...

DESCRIPTION

kwic rotates lines from the input files so that each word in the sentence appears at the beginning of a line, with a special character marking the original position of the end of the line. The output from kwic is typically sorted with 'sort' and then unrotated with 'unrot' to produce a keyword-in-context index.

If no input files are given, or if the filename '-' appears, lines will be read from standard input.

FILES

SEE ALSO

unrot; sort

DIAGNOSTICS

A message is printed if an input file cannot be opened; further processing is terminated.

AUTHORS

Original from Kernighan and Plauger's 'Software Tools', with modifications by Debbie Scherrer.

BUGS/DEFICIENCIES

NAME

Lam - laminate files

SYNOPSIS

```
lam { -string | file } ...
```

DESCRIPTION

Lam laminates the named files to the standard output. That is, the first output line is the result of concatenating the first lines of each file, and so on. If the files are different lengths, null lines are used for the missing lines in the shorter files.

The "-string" arguments are used to place strings in each output line. Each "string" is placed in the output lines at the point it appears in the argument list. For example,

```
lam -file1: foo1 "-, file2:" foo2
```

results in output lines that look like

```
file1: a line from foo1, file2: a line from foo2
```

The escape sequences described in find (and change) are valid in "string" arguments. Thus

```
lam foo1 -@n foo2
```

results in the lines from foo1 and foo2 being interleaved.

Files and string specifications may appear in any order in the argument list.

If no file arguments are given, or if the file "-" is specified, lam reads the standard input.

FILES

none

SEE ALSO

comm, tail

DIAGNOSTICS

too many arguments

The maximum number of command line arguments allowed has been exceeded. It is set by the MAXARGS definition in the source code.

too many strings

The max number of characters in a string has been exceeded.

It is set by the MAXBUF definition in the source code.

output buffer exceeded
The size of the output line buffer has been exceeded. It is set by the MAXOBUF definition in the source code.

AUTHORS

David Hanson and friends (U. of Arizona)

BUGS/DEFICIENCIES

NAME
    Lcnt - line count

SYNOPSIS
    lcnt [file] ...

DESCRIPTION
    lcnt counts the number of lines of text in the named input files, or the standard input if no files are given or the filename '-' appears. A line is zero or more characters terminated by a NEWLINE marker.

    lcnt could also be implemented as a shell script file:

```
tr '!@n' | ccnt
```

FILES

SEE ALSO
    ccnt; wcnt; the Unix command 'wc'

DIAGNOSTICS
    A message is printed if an input file could not be opened; processing is terminated.

AUTHORS
    Original from Kernighan and Plauger's 'Software Tools', with modifications by Debbie Scherrer.

BUGS/DEFICIENCIES

NAME
    Ld - loader

SYNOPSIS
    ld [-dmv] [-l[libname]] [-ptaskname] [-xs] name ...

DESCRIPTION
ld links together the named modules in the order given, searches the system libraries to resolve global references and generates an executable process.

ld understands the following flags:

-d causes 'ld' to do whatever is necessary to incorporate a system-specific debugger into the image.

-l signifies that the filename concatenated to the flag is a library name. -l alone stands for the ratfor system library, 'rlib'. The default extension for a library file is '.olb'. A library is searched when its name is encountered, so the placement of -l is significant. If the ratfor system library is not explicitly mentioned, it is searched after all other files have been linked. The fortran system library is searched at the very end.

-m causes 'ld' to do whatever is necessary to generate a system-specific load map.

-p signifies that the file name concatenated to the flag is to be the process name. If this option is not specified, the process name is determined in one of two ways:

1. The first non-library file name (eg. format.obj) is found, and the file's extension is replaced by '.exe' (format.exe). This is then the resulting process name.

2. Failing 1 (implying that all files listed in the argument list are libraries), the process image is placed on the file a.out, overwriting the previous contents of that file.

-v verbose option; output additional information about the loading process.

-x operating system specific loader options are appended to the '-x' flag. Legal sub-obtions are:

s indicates that linkage to the ratfor shared library image RLIBSHARE is NOT to be performed. This permits images to be generated for use on systems where the shared library image will not reside, as well as permitting debugging of new versions of the routines which are in the shared library image.

SEE ALSO

rc, fc

AUTHORS

Joe Sventek wrote the interface of ld to the DEC linker.

BUGS/DEFICIENCIES

NAME
    Ll - print line lengths

SYNOPSIS
    ll [file] ...

DESCRIPTION
    ll prints the lengths of the shortest and longest lines in the named files. The name "-" may be used to refer to the standard input. If no files are given, ll reads the standard input.

    NEWLINE characters are not counted as part of the length of a line.

FILES
    none

DIAGNOSTICS
    A message is issued if a named file could not be opened.

AUTHORS
    David Hanson and friends (U. of Arizona)

BUGS/DEFICIENCIES

NAME

Lpr - queue file to printer

SYNOPSIS

```
lpr [-n] [-l] [-v] [-c<num of copies>] [file]...
```

DESCRIPTION

lpr takes the named files (or standard input if none are specified) and queues copies of them to the printer. All overstriking and underlining in the documents which have been achieved via backspaces are converted to the appropriate overstrike lines to drive the printer. The switches have the following meaning:

-n	narrow paper queue. These files are queued to the printer with forms=1.
-l	label queue. These files are queued to the printer with forms=6.
-v	verbose. The job number of the print job will be displayed on the screen at the successful queueing of the file.
-cn	number of copies. The number of copies of the file to be queued may be specified this way. The default is 1 copy.

The default behavior of lpr is to queue the files with forms=0. In all cases, the print queue to which the symbiont messages are directed is sys$print. If you do not maintain this queue, you will have to modify the source code for lpr. The routine to change is lpr.w/lpr.r/dispoz.

FILES

SEE ALSO

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME

Ls - list contents of directory

SYNOPSIS

```
ls [-1dhnrtv] [-fstring] [pathname] ...
```

DESCRIPTION

Ls lists information about each file argument. When no argument is given, the default directory is listed. The file arguments may include any of the legal regular expressions described in the man entry for the editor, with the added feature that the comparisons will be case insensitive. By default, the files are listed in the order in which they are found in the directory. There are seven options:

-1 force single column output to the terminal. The default is multi-column output to the terminal, single to a disk file.
-d print only directory files found in this directory
-h print a header at the top of verbose listings
-n sort the directory by name
-v list in verbose format
-t sort by time modified (oldest first)
-r reverse the sense of the sort

-f use ‘string’ to specify the output format as follows:

b	size of file in blocks (normally 512 characters)
c	size of file in characters
m	modification date and time (dd-mmm-yy hh:mm:ss)
n	filename
o	file owner’s username
p	protection codes (oooo|gggg|wwww)
t	file type (asc|bin|dir)

The ‘b’, ‘c’, ‘n’ and ‘o’ options accept an integer prefix which specifies the field width to be used.

The verbose option formats its output as if you had specified "-f17n p m 6b o" as a format string.

It is necessary to surround the string (including the ‘-f’) with quotes if it contains any BLANKs or TABs.

EXAMPLES

The following command will cause all of the files which contain the string tst anywhere in the file name to be deleted:

```
% ls tst | args rm
```

FILES

lstemp1, lstemp2

AUTHORS

Ls was written by Joe Sventek. The '-f' option was added by Dave Martin.

SEE ALSO

ed - text editor for description of regular expressions
args - argument exploder
d - directory lister (with different default format)
fd - fast directory lister in sort order

NAME

Macro - process macro definitions

SYNOPSIS

```
macro [file] ...
```

DESCRIPTION

Macro reads the source file(s) and writes onto the standard output a new file with the macro definitions deleted and the macro references expanded. If no file names are specified, the standard input is read.

Macros are generally used to extend some underlying language to perform a translation from one language to another; that is, a macro processor allows one to define symbolic constants so that subsequent occurrences of the constant are replaced by the defining string of characters. The general format is:

```
define(name,replacement text)
```

All subsequent occurrences of "name" in the file will be replaced by "replacement text". Blanks are significant and may occur only inside the replacement text. Upper and lower case letters are also significant. Nesting of definitions is allowed, as is recursion. The definition may be more than one line long.

An elementary example of a macro is:

```
define(EOF,-1)
```

Thereafter, all occurrences of "EOF" in the file would be replaced by "-1".

Macros with arguments may also be specified. Any occurrence in the replacement text of "$n", where n is between 1 and 9, will be replaced by the nth argument when the macro is actually called. For example,

```
define(copen,$3 = open($1,$2)
             if ($3 == ERR)
                  call cant($1))
```

would define a macro which, when called by "copen(name, READ, fd)" would expand into:

```
fd = open(name,READ)
if (fd == ERR)
     call cant(name)
```

If a macro definition asks for an argument that wasn't supplied, the "$n" will be ignored.

Macros can be nested, and any macros encountered during argument collection are expanded immediately--unless they are surrounded by brackets "[]". That is, any input surrounded by [and] is left absolutely alone, except that one level of [and] is stripped off. Thus it is possible to write the macro "d" as

```
define(d,[define($1,$2)])
```

The replacement text for "d", protected by the brackets is literally "define($1,$2)" so one could say

```
d(a,bc)
```

and be assured that "a" would be defined to be "bc". Brackets must also be used when it is desired to redefine an identifier:

```
define(x,y)
define(x,z)
```

would define "y" in the second line, instead of redefining "x". To avoid redefining "y", the operation must be expressed as

```
define(x,y)
define([x],z)
```

The macro processor also includes a conditional test, with the built-in function "ifelse". The input

```
ifelse(a,b,c,d)
```

compares "a" and "b" as character strings. If they are the same, "c" is pushed back onto the input; if they differ, "d" is pushed back. As a simple example,

```
define(compare,[ifelse($1,$2,yes,no)])
```

defines "compare" as a two-argument macro returning "yes" if its arguments are the same, and "no" if they are not. The brackets prevent the "ifelse" from being evaluated too soon.

Another built-in function available is "incr". "incr(x)" converts the string "x" to a number, adds one to it, and returns that as its replacement text (as a character string). "x" had better be numeric, or the results may be undesireable. "incr" can be used for tasks like

```
define(MAXCARD,80)
define(MAXLINE,[incr(MAXCARD)])
```

which makes two parameters with values 80 and 81.

The third built-in function available in the macro processor is a function to take substrings of strings.

```
substr(s, m, n)
```

produces the substring of "s" which starts at position "m" (with origin one), of length "n". If "n" is omitted or too big, the rest of the string is used, while if "m" is out of range the result is a null string. For example,

```
substr(abc, 2, 1)
```

results in "b",

```
substr(abc, 2)
```

results in "bc", and

```
substr(abc,4)
```

is empty.

The last built-in function available in the macro processor is one to perform simple arithmetic functions:

```
arith(operand1,op,operand2)
```

where the operation specified by 'op' may be + (add), - (subtract), * (multiply), or / (divide). Negative numbers are not handled yet. Thus,

```
define(add,[arith($1,+,$2)])
add(5,3)
```

would produce the result '8'.

As a final example, here is a macro which computes the length of a character string:

```
define(len,[ifelse($1,,0,[incr(len(substr($1,2)))])])
```

Note the recursion, which is perfectly permissible. The outer layer of brackets prevents all evaluation as the definition is being copied into an internal table. The inner layer prevents the "incr" construction from being done as the arguments of the "ifelse" are collected. The value of a macro call "len(abc)" would be 3.

FILES

none

SEE ALSO

Kernighan and Plauger's "Software Tools", pages 251-283

DIAGNOSTICS

arg stack overflow
: The maximum number of total arguments has been exceeded. Currently this is 100.

call stack overflow
: The maximum level of nesting of definitions has been exceeded. Currently this is 130.

EOF in string
: An end-of-file has been encountered before a bracketed string has been terminated.

evaluation stack overflow
: The total number of characters for name, definition, and arguments has been exceeded. Currently this is 500.

unexpected EOF
: An end-of-file was reached before the macro definition was terminated.

filename: cant open
: For some reason, the file specified could not be opened. This is an unlikely error to occur; if it does show up it probably indicates a problem with the low-level primitives being used by the system.

AUTHORS

From "Software Tools" by Kernighan and Plauger, with minor modifications by Debbie Scherrer.

BUGS/DEFICIENCIES

There can be no space between the "define" and the left-parenthesis following it.

Keywords (e.g. define, ifelse, etc.) in the input file must be surrounded by brackets if they are not part of a macro--otherwise they will be stripped out by the processor.

Likewise, if brackets are desired anywhere in the input file other than in a macro, they must be surrounded by brackets themselves.

The error messages generated by the ratfor compiler when processing macros do not seem to show up in this processor. Examples are "definition too long", "missing comma in define", and "non-alphanumeric name".

NAME
    Man - display section of users manual

SYNOPSIS
    man [-<pagelen>] [-s<section>] [-a] [name] ...

DESCRIPTION
    man locates and displays the manual entries for the particular utility or function names found in the argument list. If no names are supplied, a list of those entries known to man in the section specified is displayed. If no section name is supplied (i.e. "-s") a list of available manual sections is displayed.

The manual as delivered consists of four sections:

1 The writeups for the utilities are contained here. This is the default section if none is specified. A valid synonym for '1' is 'cmd'.

2 The writeups for the primitive functions are contained here. The primitive functions are those which represent the virtual system calls for the Software Tools Virtual Machine. A valid synonym for '2' is 'prim'.

3 The writeups for the portable library functions are found in this section. Routines for manipulating archive modules, in-memory storage, push-back stacks, pattern matching, and many others are described here. Many times a problem which you are trying to solve has been solved before, with the code for the solution appearing in the library. A valid synonym for '3' is 'lib'.

4 Primers for using various utilities and function libraries appear here. A valid synonym for '4' is 'primer'.

In addition, site-dependent sections can be added by creating the necessary known files in ~man. The section on FILES below describes the structure of the known files.

By default, man will search through all sections for an entry describing 'name'; the sections are searched in the order specified in the file '~man/mpath'. The first entry found along this search path is displayed; the remainder of the sections are scanned to see if other entries describing 'name' can be found. If more are found, a note describing the section containing the additional entry is displayed to the user.

If the -a flag is specified, all of the manual entries for the particular section are displayed on standard output. If no section is specified, this results in the display of the entire manual.

When displaying to the terminal, excess white space is removed from the entries. The output is also paged when to a terminal, with the default page length being 22 lines. This value may be changed through the use of the '-<pagelen>' option. Specifying a pagelength of 0 turns the paging off. When in paging mode, the user will be asked if the next screenful of the current entry is desired. In addition, if more than one entry was requested, the user is asked if the next entry is desired.

EXAMPLES

To get a listing of all manual sections available:

```
                man -s
```

To get a listing of all entries in section (1):

```
                man -s1
```

To get the entry for the "format" utility:

```
                man format
```

To get the primer for the "ed" text editor:

```
                man -sprimer ed
```

To get the entry for the library routines "scopy" and "strcpy":

```
                man scopy strcpy
```

FILES

Accesses the known files for each section in the ~man directory.

Each section consists of two files in ~man:

* s<section-name> is an archive of the 'format' output files for each entry, with each archive module having the name of the entry. For example, s1 has the entries for the commands, with the entry for 'ar' being ar.

* i<section-name> is an index of the s-file above generated by 'asam'. This index must be sorted, and is generated by

  ```
  asam <s<section-name> | sort >i<section-name>
  ```

There is no restriction of '<section-name>' to integers, such that if one wishes to create a local man section, simply archive the formatted entries in ~man/slocal and generate the index in ~man/slocal. The section name `local' should then be added to the search path file, `~man/mpath'.

SEE ALSO
The tools 'intro' and 'apropos'; the Unix command 'man'

DIAGNOSTICS
A message is printed if the entry specified by 'name' cannot be located.

AUTHORS
Joe Sventek. The "bare -s" option was added by Dave Martin.

NAME

Mcol - multicolumn formatting

SYNOPSIS

```
mcol [-cn] [-ln] [-wn] [-gn] [-dn] [file] ...
```

DESCRIPTION

mcol reads the named files and formats them into multicolumn output on the standard output. If the filename "-" is given, or no files are specified, the standard input is read.

The options are as follows.

-cn Format the output into "n" columns. Default is 2.

-ln Set the output page size to "n". Mcol produces its output in pages, but does not place separators between the pages on the assumption that some subsequent processor will do that. (The default page length is 55.)

-wn Set the column width to "n" characters. Lines longer than "n" characters are truncated. (The default column width is 60.)

-gn Set the "gutter" width to "n". The gutter is the white space between columns. (The default gutter width is 8.)

-dn Assume output is to be printed on a display terminal. The column size is set to "n" characters and the page size is set to 24 lines. The number of columns and gutter width are computed to maximize the amount of information on a single screen. If "n" is omitted, 10 is used, which is useful for displaying lists of file names.

FILES

none

SEE ALSO

DIAGNOSTICS

invalid column count
invalid page size
invalid column width
invalid gutter width
The value of one of the option flags is invalid or exceeds the limitations of mcol.

ignoring invalid flag
A command argument option flag was given which mcol didn't

recognize.

insufficient buffer space
: Mcol could not buffer an entire page. This is usually the result of options that specify a large page size or many columns. The buffer size is set by the MAXBUF definition in the source code.

too many lines
: The number of lines per page times the number of columns exceeded mcol's line buffer space. The maximum number of lines allowed is set by the MAXPTR definition in the source code.

BUGS/DEFICIENCIES

AUTHORS

Original by David Hanson and friends (U. of Arizona), with modifications by Debbie Scherrer (LBL).

NAME
    MkDir - create directories

SYNOPSIS
    mkdir dirname ...

DESCRIPTION
    MkDir creates the specified directories.

EXAMPLES

```
                              mkdir verbs
```

    would create a subdirectory named ``verbs'' in the current directory.

```
                             mkdir ~usr/src
```

    would create a subdirectory named ``src'' in directory ``~usr''.

FILES
    none

IMPLEMENTATION
    MkDir spawns the DCL "create/directory" command, with all the default options.

SEE ALSO
    The UNIX command ``mkdir''.

DIAGNOSTICS
    ? Can't spawn ``create/directory''.

AUTHORS
    Dave Martin (Hughes Aircraft)

BUGS/DEFICIENCIES
    None of the DCL options may be specified.

NAME
    Mv - move (or rename) a file

SYNOPSIS
    mv old new

DESCRIPTION
    mv changes the name of ‘old’ to ‘new’. If ‘new’ already exists, it is removed before ‘old’ is renamed. On networks or other systems where a simple rename is impossible, mv copies the file and then deletes the original.

FILES
    none

SEE ALSO
    The Unix command ’mv’

DIAGNOSTICS
    A message is printed if ‘old’ does not exist.

AUTHORS
    Joe Sventek, Debbie Scherrer

BUGS/DEFICIENCIES
    Mv may only be used with ASCII files on many systems.

NAME

Number - number lines

SYNOPSIS

```
number [-f] [-z] [-i<n>] [-s<n>] [-d<n>] [-] file ...
```

DESCRIPTION

Number copies its input to STDOUT, adding line numbers to each line. The options are:

```
  -     read input from STDIN.

  -f    (Fortran) start numbers in column 73.  Default is 1.
The number of digits is set to 8 ("-d8").

  -z    zero-fill numbers. Default is blank-fill.

  -i<n> set line number increment to <n>.

  -s<n> start numbering with <n>.

  -d<n>  make numbers <n> digits long. default is 7.
```

FILES

none

DIAGNOSTICS

none

AUTHORS

Dave Martin (Hughes Aircraft)

BUGS/DEFICIENCIES

Tabs are assumed to be 8 spaces wide starting in column 9.
The -f option assumes lines are less than 73 columns long.

NAME

Os - convert backspaces into multiple lines for "printers"

SYNOPSIS

os [file] ...

DESCRIPTION

os (overstrike) looks for backspaces in the files specified and generates a sequence of print lines with carriage control codes to reproduce the effect of the backspaces.

If no files are given, or the filename '-' appears, input is taken from the standard input.

FILES

SEE ALSO

lpr - queue file to line printer
ul - process overstrikes for "terminals"

DIAGNOSTICS

A message is printed if an input file cannot be opened; further processing is terminated.

AUTHORS

Original from Kernighan & Plauger's 'Software Tools', with modifications by Debbie Scherrer.

BUGS/DEFICIENCIES

NAME

Pack - pack words into columns

SYNOPSIS

pack [-n] [file] ...

DESCRIPTION

pack takes the words (groups of characters separated by blanks or tabs) found on the specified files (standard input if none are specified) and outputs them to standard output in columns, 16 spaces wide, ordered from left to right. The characters used to achieve the separation of columns are TAB characters, such that those terminals which support hardware tabs can be driven efficiently. By default, five (5) columns are generated; this value can be overridden through the specification of the -n switch, where n is a decimal number.

FILES

SEE ALSO

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME

Pl - print specified lines/pages in a file

SYNOPSIS

pl [-pn] numbers [file] ...

DESCRIPTION

pl prints the specified lines from each of the named files on the standard output. If no files are given, or if the name "-" is specified, pl reads the standard input.

The "numbers" argument is a list of line numbers separated by commas, e.g.

```
pl 4,5,26,55 foo bazrat
```

prints lines 4, 5, 26, and 55 in file "foo" and "bazrat". The line numbers may be given in any order. Repeated numbers cause the specified lines to be printed once for each occurrence of the line number. Line number ranges can also be given, e.g. 4-15.

The "-p" option causes pl to print pages instead of lines, and the numbers refer to page numbers. If an integer follows the "-p", it is taken as the page size; the default is 23. Repeated numbers cause the specified pages to be printed once for each occurrence of the page number.

DIAGNOSTICS

bad page size
: Invalid page size specified after '-p' flag

bad number
: Invalid number given as argument

bad range
: Invalid range given as argument

too many numbers
: Number of lines/pages specified overflowed the buffer. Maximum number of lines is determined by the MAXLINES definition in the source code.

ignoring invalid argument
: An invalid flag was specified. Processing continues.

AUTHORS

David Hanson and friends (U. of Arizona)

BUGS/DEFICIENCIES

There is a limit to the size of pages which can be buffered. This is set by the MAXBUF definition in the source code.

NAME

Pr - paginate files to standard output

SYNOPSIS

pr [-l<n>] [file] ...

DESCRIPTION

pr paginates the named files to standard output. Each file is printed as a sequence of pages. Each page is 60 lines long, including a 3-line header and no footer. This gives 57 lines of text. The default format matches the printer control used on most line printers. The header includes the file name, possibly the date, and the page number.

If the file '-' is specified, or no file names are given, the standard input is read.

Option flags include:

-l<n> Sets the page length to '<n>'. Default page length is 60.

SEE ALSO

os, detab, mcol, format, cat

DIAGNOSTICS

ignoring invalid argument
An option flag was specified which pr did not understand

A message is printed if an input file could not be opened

AUTHORS

Original from the Kernighan-Plauger 'Software Tools' book, with modifications by David Hanson and friends (U. of Arizona) and Debbie Scherrer (LBL)

BUGS/DEFICIENCIES

The header and trailer spacing can be modified by adjusting the MARGIN1, MARGIN2, and BMARGIN definitions in the source code.

NAME

Printf - justify fields of data in fixed-width fields

SYNOPSIS

printf [-t[c] | fieldlist] outputformat [file] ...

DESCRIPTION

printf is used to manipulate data kept in formatted fields. It selects data from certain fields of the input files and copies it into fixed-width fields, with justification, in the standard output.

The 'fieldlist' parameter is used to describe the interesting columns on the input file. Fields are specified by naming the columns in which they occur (e.g. 5-10) or the columns in which they start and an indication of their length (e.g. 3+2, meaning a field which starts in column 3 and spans 2 columns). When specifying more than one field, separate the specs with commas (e.g. 5-10,16,72+8) Fields may overlap, and need not be in ascending numerical order (e.g. 1-25,10,3 is OK).

If input fields do not fall in certain columns, but rather are separated by some character (such as a blank or a comma), describe the fields by using the '-tc' flag, replacing 'c' with the appropriate separator (a tab character is the default).

Once fields have been described with either the '-tc' flag or a fieldlist, they can be arranged on output by the 'outputformat' argument. This argument is actually a picture of what the output line should look like. Fields from the input are referred to as "%[-][n]s", with the following meanings for the optional characters:

n The next input field is to be output in a field 'n' characters wide, right justified in the field.

- The input field is to be left justified in the specified field.

If a percent character is to be output, it can be specified either as %% or as @%. A percent character followed by anything other than % or [-][n]s is a syntax error in the outputformat argument. For example, an outputformat of:

```
                  "%-10s is equivalent to %s"
```

would produce an output line such as:

```
               field1     is equivalent to field2
```

If no input files are specified, or if the filename '-' is found, field will read from the standard input.

If re-ordering of a set of fields before output is necessary, the ‘field’ tool can be used prior to ‘printf’:

```
field "$3@t$2@t$1" | printf "%-15s | %-15s | %s" >outfile
```

DIAGNOSTICS

Field specification error.
: The fieldlist specification was in error, probably because it contained letters or some other illegal characters

Incorrectly formatted string.
: The outputformat specification contains an illegal % contruct.

Too many fields for internal storage.
: The fieldlist specification or the outputformat specification provides for more fields than internal storage can handle. The program can be recompiled with a larger value for the symbol MAXFIELDS.

SEE ALSO

sedit(1), field(1)

AUTHORS

Joe Sventek

NAME
    Prlabl - format labels for printing

SYNOPSIS
    prlabl [-width] <label_file

DESCRIPTION
    'prlabl' formats addresses (or other block data) for printing on sticky label forms. The default behavior assumes that each label is 9 lines wide, which corresponds to 1.5 inch labels on a 6 pitch printer or terminal. If the '-width' option is specified, 'width' is taken to be the number of lines per label. The code forces a blank line on either side of each block of data, thus limiting the data blocks to ({width | 9} - 2) lines. If a particular data block contains more than this limit, the extra lines are discarded. The data block will be centered in the window.

    The format of the address files is quite simple: all contiguous non-blank lines between blank lines are considered a single block. Any lines in the block which start with the character '#' are considered to be comments, and excluded from the block when printing.

FILES

SEE ALSO

DIAGNOSTICS

AUTHORS
    Joe Sventek

BUGS/DEFICIENCIES

NAME

Ps - list process status information

SYNOPSIS

ps [-ahx] [-tttname] [-uusername]

DESCRIPTION

ps lists information concerning processes in the system. The processes are listed in order by process id, with all child processes appearing in heirarchical order immediately below their respective parents. The default is to list all processes active at the invoking terminal. The switches cause other information to be displayed as follows:

-a	list information on processes associated with all logged in terminals on the system
-h	place header labels above the columns of information
-x	list information on all processes in the system
-tname	list information on all processes associated with terminals which contain the pattern 'name'.
-uname	list information on all processes owned by users whose names contain the pattern 'name'.

The display consists of the following information:

1. The terminal name
2. The owning user name
3. The process name
4. The de-noised image name being run by the process, or blank for DCL.
5. The total elapsed CPU time of the process
6. The process id

FILES

SEE ALSO

who - who is on the system

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME
    Pstat - determine status of process

SYNOPSIS
    pstat processid [processid ...]

DESCRIPTION
    pstat determines the status of processes specified by the processid's in the command line. It returns either active or completed for each process. The processid's are those returned by the shell when a background process is spawned.

FILES
    none

SEE ALSO
    sh - shell (command line interpreter)

DIAGNOSTICS

AUTHORS
    Joe Sventek (VAX)

BUGS/DEFICIENCIES

NAME
Pwd - print working directory name on standard output

SYNOPSIS
```
pwd [-l]
```

DESCRIPTION
pwd prints the pathname of the working (current default) directory. If the -l switch is present, the current working directory is printed out in the local parlance. This path name is of the form

```
/device/directory
```

For example,

```
                           /u/usrlib
```

is equivalent to

```
                           u:[usrlib]
```

on VMS

SEE ALSO
cd - change working directory

DIAGNOSTICS

AUTHORS
Joe Sventek

BUGS/DEFICIENCIES

NAME
    Rar - rearrange archive

SYNOPSIS
    rar [-cv] archive

DESCRIPTION
    'rar' permits the rearrangement of the modules of an archive, 'archive'. 'rar' opens 'archive' and notes the names and starting address of each module. It then reads the names of modules from standard input and outputs each module so indicated to standard output. Upon detecting an EOF on standard input, any modules not yet output are written out in the order found in the original archive.

    Switches:

    -c Suppresses the output of modules not specified on the standard input. This permits the selection of only a subset of the original archive's modules.

    -v Print the name of each module on error output after it has been successfully output to the standard output.

    Example of use:

    Suppose that you wish to create a new version (newarch) of an archive (oldarch) with all of the modules sorted by name. The following shell command will suffice:

```
ar t oldarch | sort | rar -v oldarch >newarch
```

FILES
    none

SEE ALSO
    ar - archive file maintainer

DIAGNOSTICS

AUTHORS
    Joe Sventek

BUGS/DEFICIENCIES

NAME

Ratfor - RatFor preprocessor

SYNOPSIS

```
ratp1 [-n] [file] ... | ratp2 >outfile

ratfor [-n] [file] ... >outfile

rat77 [-n] [file] ... >outfile
```

DESCRIPTION

Ratfor translates the ratfor programs in the named files into Fortran. If no input files are given, or the filename '-' appears, the standard input will be read.

Unless the '-n' flag has been specified, a file containing general purpose software tools definitions (e.g. EOF, EOS, etc.) will be automatically opened and processed before any of the files specified are read.

Syntax:

Ratfor has the following syntax:

```
   prog:   stmt
           prog stmt
   stmt:   if (expr) stmt
           if (expr) stmt else stmt
           while (expr) stmt
           repeat stmt
           repeat stmt until (expr)
           for (init clause; test expr; incr clause) stmt
           do expr stmt
           do n expr stmt
           break
           break n
           next
           next n
           return (expr)
           switch (expr) {
             case expr: stmt
             ...
             default: stmt
             }
           digits stmt
           { prog }  or  [ prog ]
           other
   other:  anything unrecognizable (i.e. fortran)
   clause: other
           clause, other
```

where 'stmt' is any Fortran or Ratfor statement. A statement is terminated by an end-of-line or a semicolon.

Character Translation:

The following character translations are performed:

<	.lt.				
<=	.le.				
==	.eq.				
!=	.ne.	^=	.ne.	~=	.ne.
>=	.ge.				
>	.gt.				
|	.or.				
&	.and.				
!	.not.	^	.not.	~	.not.

Included files:

The statement

```
               include file        or
               include "file"
```

will insert the contents of the specified file into the ratfor input in place of the 'include' statement. Quotes must surround the file name if it contains characters other than alphanumerics or underscores.

Macro Definitions:

The statement

```
               define(name,replacement text)
```

defines 'name' as a macro which will be replaced with the indicated text when encountered in the source files. Any occurrences of the strings '$n' in the replacement text, where 1 <= n <= 9, will be replaced with the nth argument when the macro is actually invoked. For example:

```
               define(bump, $1 = $1 + 1)
```

will cause the source line

```
               bump(i)
```

to be expanded into

```
          i = i + 1
```

The names of macros may contain letters, digits and underline characters, but must start with a letter. Upper case is not equivalent to lower case in macro names.

The replacement text is copied directly into the lookup table with no intepretation of the arguments, which differs from the procedure used in the macro utility. This "deferred evaluation" has the effect of eliminating the need for bracketing strings to get them through the macro processor unchanged. A side effect of the deferred evaluation is that defined names cannot be forced through the processor - i.e. the string "define" will never be output from the preprocessor. The inequivalence of upper and lower case in macro names may be used in this case to force the name of a user defined macro onto the output - i.e. if the user has defined a macro named mymac, the replacement text may contain the string MYMAC, which is not defined, and will pass through the processor.

(For compatibility, an "mdefine" macro call has been included which interprets definitions before stacking them, as does the macro tool. When using this version, use "$(" and "$)" to indicate deferred evaluation, rather than the "[" and "]" used by the macro tool.)

In addition to define, several other built-in macros are provided:

arith(x,op,y)	performs the "integer" arithmetic specified by op (+,-,*,/,**) on the two numeric operands and returns the result as its replacement.
incr(x)	converts the string x to a number, adds one to it, and returns the value as its replacement (as a character string).
ifelse(a,b,c,d)	compares a and b as character strings; if they are the same, c is pushed back onto the input, else d is pushed back.
substr(s,m,n)	produces the substring of s which starts at position m (with origin one), of length n. If n is omitted or too big, the rest of the string is used, while if m is out of range the result is a null string.
lentok(str)	pushes the length of the argument (# of characters) onto the input as a character string.
undefine(sym)	removes the definition for the symbol ‘sym’, if it is defined.

Note: the statement

```
          define name text
```

may also be used, but will not always perform correctly for macros with parameters or multi-line replacement text. The functional form is preferred.

Conditional Preprocessing:

The statements

```
ifdef(macro)                      ifnotdef(macro)
     .                                  .
     .                                  .
     .                                  .
elsedef                           elsedef
     .                                  .
     .                                  .
     .                                  .
enddef                            enddef
```

conditionalize the preprocessing upon whether the macro has been previously defined or not. The ‘elsedef’ portions of the conditionals may be omitted, if desired. The conditional bodies may be nested, up to 10 levels deep.

String Declarations:

The statements

```
string name "character string"        or
string name(size) "character string"
```

declare ’name’ to be a character array long enough to accomodate the ascii codes for the given character string, one per array element. The array is then filled by data statements. The last word of ’name’ is initialized to the symbolic parameter EOS, and indicates the end of a string. EOS must be defined either in the standard definitions file or by the user. If a size is given, name is declared to be a character array of ’size’ elements. The normal escape sequences are supported in strings; in addition, to embed a quote (") in the string, one must type @".

String Literals:

The processing of in-line quoted strings ("..." appearing outside of the scope of a 'string' declaration) is dependent upon which version of the processor you are using:

ratfor "str" is converted to 3Hstr. This action is identical to previous versions of the pre-processor.

ratp1 "str" is converted to an appropriate declaration for a 'character' array, and the appropriate data statements are output. The variable name will be of the form STNNNZ, where NNN is replaced by a rotating sequence number. The array will be declared long enough to place the value of EOS in the last element, just as for the 'string' declaration. Since these declarations are output immediately, the resulting FORTRAN code must be run through the program 'ratp2', which will reorder the code to be ANSI-66 compliant.

rat77 "str" is converted to the FORTRAN-77 constant 'str'. It is expected that this version of the preprocessor will NOT automatically load the standard symbols file, thus permitting the use of 'rat77' to preprocess F77 code.

Regardless of the version used, string literals can be continued across line boundaries by ending the line to be continued with an underline. The underline is not included as part of the literal. Leading blanks and tabs on the next line are ignored. If a quote (") is to be embedded in the string, it must be escaped, as in

```
"a quote (@") in a string"
```

In addition, the normal escape sequences are supported in the 'ratp1' version.

Character Literals:

Character constants of the form 'c' are converted to the decimal integer representation of that character in the ASCII character set. For example:

```
     call putc('!')
```

would become

```
     call putc(33)
```

The normal escape characters are supported as character constants. For example

```
     '@n'
```

is a NEWLINE (10).

Note that this capability pre-empts the use of apostrophes for delimiting string literals. Attempts to pre-process programs utilitizing apostrophes for string literals will generate syntax errors of the form:

```
            missing apostrophe in character literal
```

An utility `ratfix' is available for quickly correcting such code.

Integer Constants:

Integer constants in bases other than decimal may be specified as n%dddd... where 'n' is a decimal number indicating the base and 'dddd...' are digits in that base. For bases > 10, letters are used for digits above 9. Examples include: 8%77 (=63), 16%2ff (=767), 2%0010011 (=19). The number is converted to the equivalent decimal value using multiplication; this may cause sign problems if the number has too many digits.

Lines and Continuation:

Input is free-format; that is, statements may appear anywhere on a line, and the end of the line is generally considered the end of the statement. However, lines ending in special characters such as comma, +, -, and * are assumed to be continued on the next line. An exception to this rule is within a condition; the line is assumed to be continued if the condition does not fit on one line. Explicit continuation is indicated by ending a line with an underline character (_). The underline character is not copied to the output file.

Comments:

Comments are preceded by '#' signs and may appear anywhere in the code.

Literal (unprocessed) Lines:

Lines can be passed through ratfor without being processed by putting a percent "%" as the first character on the line. The percent will be removed and the line shifted one position to the left, but otherwise will be output without change. Macro invocations, long names, etc., appearing in the line will not be processed.

Literal (unprocessed) Character Sequences:

Sequences of characters can be passed through the processor, thus avoiding processing, by surrounding then with the tokens %(...%). The surrounding %[()] tokens will be removed and the character sequence will be output without change. Macro invocations, long names, etc. appearing in the character sequence will NOT be processed.

Long Variable Name Processing:

An optional capability available in the pre-processor, which may be enabled by your local tools support individual, is the capability of converting long variable names (those consisting of more than 6 alpha-numerics, embedded underscores, or both) to 6 character ANSI-66 compliant variable names. If this option is available, and has been used in a pre-processing run, a sequence of FORTRAN comment statements are output at the end of the generated FORTRAN code, with the mapping of long names to generated names.

It should be noted that this mapping is not deterministic across separate compilations; as such, if ‘get_next_input’ is compiled and placed in a library, source invocations of ‘get_next_input’ would not map into the identical 6-character name. To permit users to preload the long name table with the names of external routines, the ‘linkage’ statement may be used:

```
                linkage long_name external_name
```

The pair of names is entered into the table of known long variable names, preventing any generated names for local long variables from colliding with the external name. The programmer must provide accurate information via this statement to permit access to routines with "long variable names" across compilations.

If long variable name processing has not been enabled for your

site, linkage is synonymous with define.

NOTE: since long variable name processing is optional, its use will generate code that is inherently non-portable to sites not desiring this capability. Users wishing to write portable code should avoid long variable names.

CHANGES

This ratfor preprocessor differs from the original (as released by Kernighan and Plauger) in the following ways:

The code has been rewritten and reorganized.

Hash tables have been added for increased efficiency in searching for macro definitions and Ratfor keywords.

The 'string' declaration has been included.

The define processor has been augmented to support macros with arguments.

Conditional preprocessing upon the definition (or lack therof) of a symbol has been included.

Many extraneous gotos have been avoided.

Blanks have been included in the output for increased readability.

Multi-level 'break' and 'next' statements have been included.

The Fortran 'DO' is allowed, as well as the ratfor one.

The capability of specifying integer constants in bases other than decimal has been added.

Underscores have been allowed in names.

The 'define' syntax has been expanded to include the form: define name value

The 'return(value)' feature has been added.

Quoted file names following 'include' statements have been added to allow for special characters in file names.

A method for allowing lines to pass through un-processed has

been added.

The 'switch' control statement has been included.

Continuation lines have been implemented.

Brackets have been allowed to replace braces (but NOT '$(' and '$)')

Character constants are now supported.

Groups of FORTRAN statements are permitted in the init and re-init clauses of the for statement.

A method for allowing character sequences to pass through un-processed has been added.

An `undefine' command has been added to permit removal of symbol definitions.

Three types of literal character string processing are now possible. The default action permanently eliminates the usage of Hollerith constants in portable tools.

Long variable names processing can now be enabled as a site-dependent option.

FILES

A generalized definition file (e.g. 'ratdef') is automatically opened and read.

SEE ALSO

Kernighan and Plauger's "Software Tools"
Kernighan's "RATFOR - A Preprocessor for a Rational Fortran"
The Unix command rc in the Unix Manual
The tools 'incl' and 'macro'

DIAGNOSTICS

(The errors marked with asterisk '*' are fatal; all others are simply warning messages.)

* arg stack overflow
    The argument stack for the macro processor has been exceeded. The size of the stack is determined by the symbol ARGSIZE in the source definitions file.
o arith error
    An error occurred while evaluating the built-in macro, `arith'.
* buffer overflow

One of the preprocessor's internal buffers overflowed, possibly, but not necessarily, because the string buffers were exceeded. The definition SBUFSIZE in the preprocessor symbols file determines the size of the string buffers.

* call stack overflow
: The call stack (used to store call frames) in the macro processor has been exceeded. The definition CALLSIZE in the source definition file determines the size of this stack.

* cannot make identifier unique
: All attempts to generate an unique short variable name for the long variable name being processed failed. This message will only be seen if the long variable name processing has been enabled.

o cannot open standard definitions file
: The special file containing general purpose ratfor definitions could not be opened, possibly because it did not exist or the user did not have access to the directory on which it resides.

o can't open include
: File to be included could not be located, the user did not have privilege to access it, or the file could not be opened due to some problem in the local primitives.

o conditional processing still active at EOF
: A sufficient number of "enddef" directives have not been encountered before detecting EOF on the input file.

* Conditionals nested too deeply
: The stack for nested conditionals has overflowed. The size of the stack is specified by the value of COND_STACK_DEPTH defined in the preprocessor symbols file.

* definition too long
: The number of characters in the name to be defined exceeded Ratfor's internal array size. The size is defined by the MAXTOK definition in the preprocessor symbols file.

o duplicate case label
: Two case labels with identical values were detected.

* EOF in string
: The macro processor detected an EOF in the current input file while evaluating a macro.

* evaluation stack overflow
: The evaluation stack for the macro processor has been exceeded. This stack's size is determined by the symbol EVALSIZE in the source definition file.

* for clause too long
: The internal buffer used to hold the clauses for the 'for' statement was exceeded. Size of this buffer is determined by the MAXFORSTK definition in the preprocessor symbols

```
            file.
      * getdef is confused
            There  were  horrendous  problems when attempting to access
            the definition table
      o illegal break
            Break did not occur  inside  a  valid  "while",  "for",  or
            "repeat" loop
      o illegal case or default
            A  "case" or "default" statement was detected which was not
            in the scope of a "switch" statement.
      o illegal case syntax
            The case label  was  not  of  the  correct  form.   It  may
            consist   of   comma-separated   constants   or  ranges  of
            constants.
      o illegal else
            Else clause probably did not follow an "if" clause
      * Illegal enddef encountered
            An "enddef" directive  was  encountered  while  conditional
            preprocessing was inactive.
      o illegal next
            "Next"  did  not  occur  inside  a valid "for", "while", or
            "repeat" loop
      o illegal range in case label
            A case label specifying a range  of  values  (of  the  form
            m-n) was detected in which m > n.
      o illegal right brace
            A right brace was found without a matching left brace
      o in entdef: no room for new definition
            There  is  insufficient  memory for macro definitions, etc.
            Increase the MEMSIZE definition in the preprocessor.
      o includes nested too deeply
            There is a limit  to  the  level  of  nesting  of  included
            files.   It  is dependent upon the maximum number of opened
            files  allowed  at  a  time,  and  is  set  by  the  NFILES
            definition in the preprocessor symbols file.
      o invalid case label
            The  upper  limit  of  a  case label specifying a range was
            non-numeric.
      * invalid conditional token
            The  token  given  as  the  argument  to  an   "ifdef"   or
            "ifnotdef" directive was not alpha-numeric.
      o invalid for clause
            The  "for"  clause did not contain a valid init, condition,
            and/or increment section
      o invalid string size
            The string format 'string name(size) "..."' was  used,  but
            the size was given improperly.
      * missing `(' in conditional
            The   first   non-blank   token  following  an  "ifdef"  or
            "ifnotdef" directive was NOT a left parenthesis.
```

* missing ‘)’ in conditional
    An "ifdef" of "ifnotdef" directive was not properly terminated with a right parenthesis.
* missing ‘)’ in define
    A define(...) was not properly terminated with a right parenthesis.
* missing ‘(’ in undefine
    The first non-blank token following an "undefine" was NOT a left parenthesis.
* missing ‘)’ in undefine
    An "undefine" directive was not properly terminated with a right parenthesis.
o missing apostrophe in character literal
    An apostrophe-delimited string NOT of the form ’c’ or ’@c’ was encountered.
* missing colon in case or default label
    The list of case labels, or the default label were not followed by a colon.
* missing comma in define
    Definitions of the form ’define(name,defn)’ must include the comma as a separator.
o missing function name
    There was an error in declaring a function
o missing left brace in switch statement
    The left brace indicating the start of the block of case labels for the "switch" statement was not encountered.
o missing left paren
    A parenthesis was expected, probably in an "if" statement, but not found
o missing literal quote
    The terminating "%)" to a literally quoted string was not found.
o missing parenthesis in condition
    A right parenthesis was expected, probably in an "if" statement, but not found
o missing quote
    A quoted string was not terminated by a quote
o missing right paren
    A right parenthesis was expected in a Fortran (as opposed to Ratfor) statement but not found
o missing string token
    No array name was given when declaring a string variable
* multiple defaults in switch statement
    More than one "default" statements were detected in the scope of a single "switch" statement.
o No room for generated variable name
    The table space used for generated long variable names has been exhausted. Increase the MEMSIZE definition in the preprocessor. This message cannot appear unless the long variable name processing has been enabled.

- o No room for linkage external name
  The table space used for generated external names has been exhausted. Increase the MEMSIZE definition in the preprocessor. This message cannot appear unless the long variable name processing has been enabled.
- * non-alphanumeric name
  Definitions may contain only alphanumeric characters and underscores.
- * stack overflow in parser
  Statements were nested at too deep a level. The stack depth is set by the MAXSTACK definition in the preprocessor symbols file.
- * switch table overflow
  More case labels were specified than the internal storage can handle. The size of the internal storage is determined by the value of MAXSWITCH defined in the preprocessor symbols file.
- o token too long
  A token (word) in the source code was too long to fit into one of Ratfor's internal arrays. The maximum size is set by the MAXTOK definition in the preprocessor symbols file.
- * too many characters pushed back
  The source code has illegally specified a Ratfor command, or has used a Ratfor keyword in an illegal manner, and the parser has attempted but failed to make sense out of it. The size of the push-back buffer is set by BUFSIZE in the preprocessor symbols file.
- o unbalanced parentheses
  Unbalanced parentheses detected in a Fortran (as opposed to Ratfor) statement
- o unexpected EOF
  An end-of-file was reached before all braces had been accounted for. This is usually caused by unmatched braces somewhere deep in the source code.
- o warning: possible label conflict
  This message is printed when the user has labeled a statement with a label in the 23000-23999 range. Ratfor statements are assigned in this range and a user-defined one may conflict with a Ratfor-generated one.
- * "file": cannot open
  Ratfor could not open an input file specified by the user on the command line.

AUTHORS

Original by B. Kernighan and P. J. Plauger, with rewrites and enhancements by David Hanson and friends (U. of Arizona), Joe Sventek and Debbie Scherrer (Lawrence Berkeley Laboratory), and Allen Akin (Georgia Institute of Technology).

BUGS/DEFICIENCIES

Missing parentheses or braces may cause erratic behavior. Eventually Ratfor should be taught to terminate parenthesis/brace checking at the end of each subroutine.

Although one bug was fixed which caused line numbers in error messages to be incorrect, they still aren't quite right. (newlines in macro text are difficult to handle properly). Use them only as a general area in which to look for errors.

Extraneous 'continue' statements are generated within Fortran 'do' statements. The 'next' statement does not work properly when used within Fortran 'do' statements.

There is no way to explicitly cause a statement to begin in column 6 (i.e. a Fortran continued statement), although implicit continuation is performed.

Ratfor is very slow, principally in the lexical analysis, character input, and macro processing routines (in that order). Attempts to speed it up should concentrate on the routines 'gtok', 'ngetch', and 'deftok'. An even better approach would be to re-work the lexical analyzer and parser completely.

NAME

Ratfor - RatFor preprocessor

SYNOPSIS

```
ratp1 [-n] [file] ... | ratp2 >outfile

ratfor [-n] [file] ... >outfile

rat77 [-n] [file] ... >outfile
```

DESCRIPTION

Ratfor translates the ratfor programs in the named files into Fortran. If no input files are given, or the filename '-' appears, the standard input will be read.

Unless the '-n' flag has been specified, a file containing general purpose software tools definitions (e.g. EOF, EOS, etc.) will be automatically opened and processed before any of the files specified are read.

Syntax:

Ratfor has the following syntax:

```
   prog:   stmt
           prog stmt
   stmt:   if (expr) stmt
           if (expr) stmt else stmt
           while (expr) stmt
           repeat stmt
           repeat stmt until (expr)
           for (init clause; test expr; incr clause) stmt
           do expr stmt
           do n expr stmt
           break
           break n
           next
           next n
           return (expr)
           switch (expr) {
             case expr: stmt
             ...
             default: stmt
             }
           digits stmt
           { prog }  or  [ prog ]
           other
   other:  anything unrecognizable (i.e. fortran)
   clause: other
           clause, other
```

where 'stmt' is any Fortran or Ratfor statement. A statement is terminated by an end-of-line or a semicolon.

Character Translation:

The following character translations are performed:

<	.lt.				
<=	.le.				
==	.eq.				
!=	.ne.	^=	.ne.	˜=	.ne.
>=	.ge.				
>	.gt.				
|	.or.				
&	.and.				
!	.not.	^	.not.	˜	.not.

Included files:

The statement

```
include file        or
include "file"
```

will insert the contents of the specified file into the ratfor input in place of the 'include' statement. Quotes must surround the file name if it contains characters other than alphanumerics or underscores.

Macro Definitions:

The statement

```
define(name,replacement text)
```

defines 'name' as a macro which will be replaced with the indicated text when encountered in the source files. Any occurrences of the strings '$n' in the replacement text, where 1 <= n <= 9, will be replaced with the nth argument when the macro is actually invoked. For example:

```
define(bump, $1 = $1 + 1)
```

will cause the source line

```
bump(i)
```

to be expanded into

```
        i = i + 1
```

The names of macros may contain letters, digits and underline characters, but must start with a letter. Upper case is not equivalent to lower case in macro names.

The replacement text is copied directly into the lookup table with no intepretation of the arguments, which differs from the procedure used in the macro utility. This "deferred evaluation" has the effect of eliminating the need for bracketing strings to get them through the macro processor unchanged. A side effect of the deferred evaluation is that defined names cannot be forced through the processor - i.e. the string "define" will never be output from the preprocessor. The inequivalence of upper and lower case in macro names may be used in this case to force the name of a user defined macro onto the output - i.e. if the user has defined a macro named mymac, the replacement text may contain the string MYMAC, which is not defined, and will pass through the processor.

(For compatibility, an "mdefine" macro call has been included which interprets definitions before stacking them, as does the macro tool. When using this version, use "$(" and "$)" to indicate deferred evaluation, rather than the "[" and "]" used by the macro tool.)

In addition to define, several other built-in macros are provided:

arith(x,op,y)	performs the "integer" arithmetic specified by op (+,-,*,/,**) on the two numeric operands and returns the result as its replacement.
incr(x)	converts the string x to a number, adds one to it, and returns the value as its replacement (as a character string).
ifelse(a,b,c,d)	compares a and b as character strings; if they are the same, c is pushed back onto the input, else d is pushed back.
substr(s,m,n)	produces the substring of s which starts at position m (with origin one), of length n. If n is omitted or too big, the rest of the string is used, while if m is out of range the result is a null string.
lentok(str)	pushes the length of the argument (# of characters) onto the input as a character string.
undefine(sym)	removes the definition for the symbol 'sym', if it is defined.

Note: the statement

```
               define name text
```

may also be used, but will not always perform correctly for macros with parameters or multi-line replacement text. The functional form is preferred.

Conditional Preprocessing:

The statements

```
          ifdef(macro)                      ifnotdef(macro)
                .                                 .
                .                                 .
                .                                 .
          elsedef                           elsedef
                .                                 .
                .                                 .
                .                                 .
          enddef                            enddef
```

conditionalize the preprocessing upon whether the macro has been previously defined or not. The ‘elsedef’ portions of the conditionals may be omitted, if desired. The conditional bodies may be nested, up to 10 levels deep.

String Declarations:

The statements

```
          string name "character string"         or
          string name(size) "character string"
```

declare ’name’ to be a character array long enough to accomodate the ascii codes for the given character string, one per array element. The array is then filled by data statements. The last word of ’name’ is initialized to the symbolic parameter EOS, and indicates the end of a string. EOS must be defined either in the standard definitions file or by the user. If a size is given, name is declared to be a character array of ’size’ elements. The normal escape sequences are supported in strings; in addition, to embed a quote (") in the string, one must type @".

String Literals:

The processing of in-line quoted strings ("..." appearing outside of the scope of a ‘string’ declaration) is dependent upon which version of the processor you are using:

ratfor "str" is converted to 3Hstr. This action is identical to previous versions of the pre-processor.

ratp1 "str" is converted to an appropriate declaration for a ‘character’ array, and the appropriate data statements are output. The variable name will be of the form STNNNZ, where NNN is replaced by a rotating sequence number. The array will be declared long enough to place the value of EOS in the last element, just as for the ‘string’ declaration. Since these declarations are output immediately, the resulting FORTRAN code must be run through the program ‘ratp2’, which will reorder the code to be ANSI-66 compliant.

rat77 "str" is converted to the FORTRAN-77 constant ’str’. It is expected that this version of the preprocessor will NOT automatically load the standard symbols file, thus permitting the use of ‘rat77’ to preprocess F77 code.

Regardless of the version used, string literals can be continued across line boundaries by ending the line to be continued with an underline. The underline is not included as part of the literal. Leading blanks and tabs on the next line are ignored. If a quote (") is to be embedded in the string, it must be escaped, as in

```
"a quote (@") in a string"
```

In addition, the normal escape sequences are supported in the ‘ratp1’ version.

Character Literals:

Character constants of the form ’c’ are converted to the decimal integer representation of that character in the ASCII character set. For example:

```
call putc(’!’)
```

would become

```
call putc(33)
```

The normal escape characters are supported as character constants. For example

```
'@n'
```

is a NEWLINE (10).

Note that this capability pre-empts the use of apostrophes for delimiting string literals. Attempts to pre-process programs utilitizing apostrophes for string literals will generate syntax errors of the form:

```
missing apostrophe in character literal
```

An utility `ratfix' is available for quickly correcting such code.

Integer Constants:

Integer constants in bases other than decimal may be specified as n%dddd... where 'n' is a decimal number indicating the base and 'dddd...' are digits in that base. For bases > 10, letters are used for digits above 9. Examples include: 8%77 (=63), 16%2ff (=767), 2%0010011 (=19). The number is converted to the equivalent decimal value using multiplication; this may cause sign problems if the number has too many digits.

Lines and Continuation:

Input is free-format; that is, statements may appear anywhere on a line, and the end of the line is generally considered the end of the statement. However, lines ending in special characters such as comma, +, -, and * are assumed to be continued on the next line. An exception to this rule is within a condition; the line is assumed to be continued if the condition does not fit on one line. Explicit continuation is indicated by ending a line with an underline character (_). The underline character is not copied to the output file.

Comments:

Comments are preceded by '#' signs and may appear anywhere in the code.

Literal (unprocessed) Lines:

Lines can be passed through ratfor without being processed by putting a percent "%" as the first character on the line. The percent will be removed and the line shifted one position to the left, but otherwise will be output without change. Macro invocations, long names, etc., appearing in the line will not be processed.

Literal (unprocessed) Character Sequences:

Sequences of characters can be passed through the processor, thus avoiding processing, by surrounding then with the tokens %(...%). The surrounding %[()] tokens will be removed and the character sequence will be output without change. Macro invocations, long names, etc. appearing in the character sequence will NOT be processed.

Long Variable Name Processing:

An optional capability available in the pre-processor, which may be enabled by your local tools support individual, is the capability of converting long variable names (those consisting of more than 6 alpha-numerics, embedded underscores, or both) to 6 character ANSI-66 compliant variable names. If this option is available, and has been used in a pre-processing run, a sequence of FORTRAN comment statements are output at the end of the generated FORTRAN code, with the mapping of long names to generated names.

It should be noted that this mapping is not deterministic across separate compilations; as such, if 'get_next_input' is compiled and placed in a library, source invocations of 'get_next_input' would not map into the identical 6-character name. To permit users to preload the long name table with the names of external routines, the 'linkage' statement may be used:

```
linkage long_name external_name
```

The pair of names is entered into the table of known long variable names, preventing any generated names for local long variables from colliding with the external name. The programmer must provide accurate information via this statement to permit access to routines with "long variable names" across compilations.

If long variable name processing has not been enabled for your

site, linkage is synonymous with define.

NOTE: since long variable name processing is optional, its use will generate code that is inherently non-portable to sites not desiring this capability. Users wishing to write portable code should avoid long variable names.

CHANGES

This ratfor preprocessor differs from the original (as released by Kernighan and Plauger) in the following ways:

The code has been rewritten and reorganized.

Hash tables have been added for increased efficiency in searching for macro definitions and Ratfor keywords.

The ’string’ declaration has been included.

The define processor has been augmented to support macros with arguments.

Conditional preprocessing upon the definition (or lack therof) of a symbol has been included.

Many extraneous gotos have been avoided.

Blanks have been included in the output for increased readability.

Multi-level ’break’ and ’next’ statements have been included.

The Fortran ’DO’ is allowed, as well as the ratfor one.

The capability of specifying integer constants in bases other than decimal has been added.

Underscores have been allowed in names.

The ’define’ syntax has been expanded to include the form: define name value

The ’return(value)’ feature has been added.

Quoted file names following ’include’ statements have been added to allow for special characters in file names.

A method for allowing lines to pass through un-processed has

been added.

The 'switch' control statement has been included.

Continuation lines have been implemented.

Brackets have been allowed to replace braces (but NOT '$(' and '$)')

Character constants are now supported.

Groups of FORTRAN statements are permitted in the init and re-init clauses of the for statement.

A method for allowing character sequences to pass through un-processed has been added.

An 'undefine' command has been added to permit removal of symbol definitions.

Three types of literal character string processing are now possible. The default action permanently eliminates the usage of Hollerith constants in portable tools.

Long variable names processing can now be enabled as a site-dependent option.

FILES

A generalized definition file (e.g. 'ratdef') is automatically opened and read.

SEE ALSO

Kernighan and Plauger's "Software Tools"
Kernighan's "RATFOR - A Preprocessor for a Rational Fortran"
The Unix command rc in the Unix Manual
The tools 'incl' and 'macro'

DIAGNOSTICS

(The errors marked with asterisk '*' are fatal; all others are simply warning messages.)

* arg stack overflow
    The argument stack for the macro processor has been exceeded. The size of the stack is determined by the symbol ARGSIZE in the source definitions file.
o arith error
    An error occurred while evaluating the built-in macro, 'arith'.
* buffer overflow

One of the preprocessor's internal buffers overflowed, possibly, but not necessarily, because the string buffers were exceeded. The definition SBUFSIZE in the preprocessor symbols file determines the size of the string buffers.

* call stack overflow
: The call stack (used to store call frames) in the macro processor has been exceeded. The definition CALLSIZE in the source definition file determines the size of this stack.

* cannot make identifier unique
: All attempts to generate an unique short variable name for the long variable name being processed failed. This message will only be seen if the long variable name processing has been enabled.

o cannot open standard definitions file
: The special file containing general purpose ratfor definitions could not be opened, possibly because it did not exist or the user did not have access to the directory on which it resides.

o can't open include
: File to be included could not be located, the user did not have privilege to access it, or the file could not be opened due to some problem in the local primitives.

o conditional processing still active at EOF
: A sufficient number of "enddef" directives have not been encountered before detecting EOF on the input file.

* Conditionals nested too deeply
: The stack for nested conditionals has overflowed. The size of the stack is specified by the value of COND_STACK_DEPTH defined in the preprocessor symbols file.

* definition too long
: The number of characters in the name to be defined exceeded Ratfor's internal array size. The size is defined by the MAXTOK definition in the preprocessor symbols file.

o duplicate case label
: Two case labels with identical values were detected.

* EOF in string
: The macro processor detected an EOF in the current input file while evaluating a macro.

* evaluation stack overflow
: The evaluation stack for the macro processor has been exceeded. This stack's size is determined by the symbol EVALSIZE in the source definition file.

* for clause too long
: The internal buffer used to hold the clauses for the 'for' statement was exceeded. Size of this buffer is determined by the MAXFORSTK definition in the preprocessor symbols

```
          file.
     * getdef is confused
          There  were  horrendous  problems when attempting to access
          the definition table
     o illegal break
          Break did not occur  inside  a  valid  "while",  "for",  or
          "repeat" loop
     o illegal case or default
          A  "case" or "default" statement was detected which was not
          in the scope of a "switch" statement.
     o illegal case syntax
          The case label  was  not  of  the  correct  form.   It  may
          consist   of   comma-separated   constants   or  ranges  of
          constants.
     o illegal else
          Else clause probably did not follow an "if" clause
     * Illegal enddef encountered
          An "enddef" directive  was  encountered  while  conditional
          preprocessing was inactive.
     o illegal next
          "Next"  did  not  occur  inside  a valid "for", "while", or
          "repeat" loop
     o illegal range in case label
          A case label specifying a range  of  values  (of  the  form
          m-n) was detected in which m > n.
     o illegal right brace
          A right brace was found without a matching left brace
     o in entdef: no room for new definition
          There  is  insufficient  memory for macro definitions, etc.
          Increase the MEMSIZE definition in the preprocessor.
     o includes nested too deeply
          There is a limit  to  the  level  of  nesting  of  included
          files.   It  is dependent upon the maximum number of opened
          files  allowed  at  a  time,  and  is  set  by  the  NFILES
          definition in the preprocessor symbols file.
     o invalid case label
          The  upper  limit  of  a  case label specifying a range was
          non-numeric.
     * invalid conditional token
          The  token  given  as  the  argument  to  an   "ifdef"   or
          "ifnotdef" directive was not alpha-numeric.
     o invalid for clause
          The  "for"  clause did not contain a valid init, condition,
          and/or increment section
     o invalid string size
          The string format 'string name(size) "..."' was  used,  but
          the size was given improperly.
     * missing `(' in conditional
          The   first   non-blank   token  following  an  "ifdef"  or
          "ifnotdef" directive was NOT a left parenthesis.
```

```
* missing `)' in conditional
     An "ifdef" of "ifnotdef" directive was not properly
     terminated with a right parenthesis.
* missing `)' in define
     A define(...) was not properly terminated with a right
     parenthesis.
* missing `(' in undefine
     The first non-blank token following an "undefine" was NOT
     a left parenthesis.
* missing `)' in undefine
     An "undefine" directive was not properly terminated with a
     right parenthesis.
o missing apostrophe in character literal
     An apostrophe-delimited string NOT of the form 'c' or '@c'
     was encountered.
* missing colon in case or default label
     The list of case labels, or the default label were not
     followed by a colon.
* missing comma in define
     Definitions of the form 'define(name,defn)' must include
     the comma as a separator.
o missing function name
     There was an error in declaring a function
o missing left brace in switch statement
     The left brace indicating the start of the block of case
     labels for the "switch" statement was not encountered.
o missing left paren
     A parenthesis was expected, probably in an "if" statement,
     but not found
o missing literal quote
     The terminating "%)" to a literally quoted string was not
     found.
o missing parenthesis in condition
     A right parenthesis was expected, probably in an "if"
     statement, but not found
o missing quote
     A quoted string was not terminated by a quote
o missing right paren
     A right parenthesis was expected in a Fortran (as opposed
     to Ratfor) statement but not found
o missing string token
     No array name was given when declaring a string variable
* multiple defaults in switch statement
     More than one "default" statements were detected in the
     scope of a single "switch" statement.
o No room for generated variable name
     The table space used for generated long variable names has
     been exhausted. Increase the MEMSIZE definition in the
     preprocessor. This message cannot appear unless the long
     variable name processing has been enabled.
```

o No room for linkage external name
  The table space used for generated external names has been exhausted. Increase the MEMSIZE definition in the preprocessor. This message cannot appear unless the long variable name processing has been enabled.

* non-alphanumeric name
  Definitions may contain only alphanumeric characters and underscores.

* stack overflow in parser
  Statements were nested at too deep a level. The stack depth is set by the MAXSTACK definition in the preprocessor symbols file.

* switch table overflow
  More case labels were specified than the internal storage can handle. The size of the internal storage is determined by the value of MAXSWITCH defined in the preprocessor symbols file.

o token too long
  A token (word) in the source code was too long to fit into one of Ratfor's internal arrays. The maximum size is set by the MAXTOK definition in the preprocessor symbols file.

* too many characters pushed back
  The source code has illegally specified a Ratfor command, or has used a Ratfor keyword in an illegal manner, and the parser has attempted but failed to make sense out of it. The size of the push-back buffer is set by BUFSIZE in the preprocessor symbols file.

o unbalanced parentheses
  Unbalanced parentheses detected in a Fortran (as opposed to Ratfor) statement

o unexpected EOF
  An end-of-file was reached before all braces had been accounted for. This is usually caused by unmatched braces somewhere deep in the source code.

o warning: possible label conflict
  This message is printed when the user has labeled a statement with a label in the 23000-23999 range. Ratfor statements are assigned in this range and a user-defined one may conflict with a Ratfor-generated one.

* "file": cannot open
  Ratfor could not open an input file specified by the user on the command line.

AUTHORS

Original by B. Kernighan and P. J. Plauger, with rewrites and enhancements by David Hanson and friends (U. of Arizona), Joe Sventek and Debbie Scherrer (Lawrence Berkeley Laboratory), and Allen Akin (Georgia Institute of Technology).

BUGS/DEFICIENCIES

Missing parentheses or braces may cause erratic behavior. Eventually Ratfor should be taught to terminate parenthesis/brace checking at the end of each subroutine.

Although one bug was fixed which caused line numbers in error messages to be incorrect, they still aren't quite right. (newlines in macro text are difficult to handle properly). Use them only as a general area in which to look for errors.

Extraneous 'continue' statements are generated within Fortran 'do' statements. The 'next' statement does not work properly when used within Fortran 'do' statements.

There is no way to explicitly cause a statement to begin in column 6 (i.e. a Fortran continued statement), although implicit continuation is performed.

Ratfor is very slow, principally in the lexical analysis, character input, and macro processing routines (in that order). Attempts to speed it up should concentrate on the routines 'gtok', 'ngetch', and 'deftok'. An even better approach would be to re-work the lexical analyzer and parser completely.

NAME

Ratfor - RatFor preprocessor

SYNOPSIS

```
ratp1 [-n] [file] ... | ratp2 >outfile

ratfor [-n] [file] ... >outfile

rat77 [-n] [file] ... >outfile
```

DESCRIPTION

Ratfor translates the ratfor programs in the named files into Fortran. If no input files are given, or the filename '-' appears, the standard input will be read.

Unless the '-n' flag has been specified, a file containing general purpose software tools definitions (e.g. EOF, EOS, etc.) will be automatically opened and processed before any of the files specified are read.

Syntax:

Ratfor has the following syntax:

```
   prog:   stmt
           prog stmt
   stmt:   if (expr) stmt
           if (expr) stmt else stmt
           while (expr) stmt
           repeat stmt
           repeat stmt until (expr)
           for (init clause; test expr; incr clause) stmt
           do expr stmt
           do n expr stmt
           break
           break n
           next
           next n
           return (expr)
           switch (expr) {
             case expr: stmt
             ...
             default: stmt
             }
           digits stmt
           { prog }  or  [ prog ]
           other
   other:  anything unrecognizable (i.e. fortran)
   clause: other
           clause, other
```

where 'stmt' is any Fortran or Ratfor statement. A statement is terminated by an end-of-line or a semicolon.

Character Translation:

The following character translations are performed:

<	.lt.				
<=	.le.				
==	.eq.				
!=	.ne.	^=	.ne.	~=	.ne.
>=	.ge.				
>	.gt.				
|	.or.				
&	.and.				
!	.not.	^	.not.	~	.not.

Included files:

The statement

```
               include file        or
               include "file"
```

will insert the contents of the specified file into the ratfor input in place of the 'include' statement. Quotes must surround the file name if it contains characters other than alphanumerics or underscores.

Macro Definitions:

The statement

```
               define(name,replacement text)
```

defines 'name' as a macro which will be replaced with the indicated text when encountered in the source files. Any occurrences of the strings '$n' in the replacement text, where 1 <= n <= 9, will be replaced with the nth argument when the macro is actually invoked. For example:

```
               define(bump, $1 = $1 + 1)
```

will cause the source line

```
               bump(i)
```

to be expanded into

```
            i = i + 1
```

The names of macros may contain letters, digits and underline characters, but must start with a letter. Upper case is not equivalent to lower case in macro names.

The replacement text is copied directly into the lookup table with no intepretation of the arguments, which differs from the procedure used in the macro utility. This "deferred evaluation" has the effect of eliminating the need for bracketing strings to get them through the macro processor unchanged. A side effect of the deferred evaluation is that defined names cannot be forced through the processor - i.e. the string "define" will never be output from the preprocessor. The inequivalence of upper and lower case in macro names may be used in this case to force the name of a user defined macro onto the output - i.e. if the user has defined a macro named mymac, the replacement text may contain the string MYMAC, which is not defined, and will pass through the processor.

(For compatibility, an "mdefine" macro call has been included which interprets definitions before stacking them, as does the macro tool. When using this version, use "$(" and "$)" to indicate deferred evaluation, rather than the "[" and "]" used by the macro tool.)

In addition to define, several other built-in macros are provided:

arith(x,op,y)	performs the "integer" arithmetic specified by op (+,-,*,/,**) on the two numeric operands and returns the result as its replacement.
incr(x)	converts the string x to a number, adds one to it, and returns the value as its replacement (as a character string).
ifelse(a,b,c,d)	compares a and b as character strings; if they are the same, c is pushed back onto the input, else d is pushed back.
substr(s,m,n)	produces the substring of s which starts at position m (with origin one), of length n. If n is omitted or too big, the rest of the string is used, while if m is out of range the result is a null string.
lentok(str)	pushes the length of the argument (# of characters) onto the input as a character string.
undefine(sym)	removes the definition for the symbol ‘sym’, if it is defined.

Note: the statement

```
               define name text
```

may also be used, but will not always perform correctly for macros with parameters or multi-line replacement text. The functional form is preferred.

Conditional Preprocessing:

The statements

```
          ifdef(macro)                       ifnotdef(macro)
                .                                  .
                .                                  .
                .                                  .
          elsedef                            elsedef
                .                                  .
                .                                  .
                .                                  .
          enddef                             enddef
```

conditionalize the preprocessing upon whether the macro has been previously defined or not. The ‘elsedef’ portions of the conditionals may be omitted, if desired. The conditional bodies may be nested, up to 10 levels deep.

String Declarations:

The statements

```
          string name "character string"         or
          string name(size) "character string"
```

declare ’name’ to be a character array long enough to accomodate the ascii codes for the given character string, one per array element. The array is then filled by data statements. The last word of ’name’ is initialized to the symbolic parameter EOS, and indicates the end of a string. EOS must be defined either in the standard definitions file or by the user. If a size is given, name is declared to be a character array of ’size’ elements. The normal escape sequences are supported in strings; in addition, to embed a quote (") in the string, one must type @".

String Literals:

The processing of in-line quoted strings ("..." appearing outside of the scope of a ‘string’ declaration) is dependent upon which version of the processor you are using:

ratfor "str" is converted to 3Hstr. This action is identical to previous versions of the pre-processor.

ratp1 "str" is converted to an appropriate declaration for a ‘character’ array, and the appropriate data statements are output. The variable name will be of the form STNNNZ, where NNN is replaced by a rotating sequence number. The array will be declared long enough to place the value of EOS in the last element, just as for the ‘string’ declaration. Since these declarations are output immediately, the resulting FORTRAN code must be run through the program ‘ratp2’, which will reorder the code to be ANSI-66 compliant.

rat77 "str" is converted to the FORTRAN-77 constant ’str’. It is expected that this version of the preprocessor will NOT automatically load the standard symbols file, thus permitting the use of ‘rat77’ to preprocess F77 code.

Regardless of the version used, string literals can be continued across line boundaries by ending the line to be continued with an underline. The underline is not included as part of the literal. Leading blanks and tabs on the next line are ignored. If a quote (") is to be embedded in the string, it must be escaped, as in

```
                    "a quote (@") in a string"
```

In addition, the normal escape sequences are supported in the ‘ratp1’ version.

Character Literals:

Character constants of the form ’c’ are converted to the decimal integer representation of that character in the ASCII character set. For example:

```
     call putc(’!’)
```

would become

```
     call putc(33)
```

The normal escape characters are supported as character constants. For example

```
     '@n'
```

is a NEWLINE (10).

Note that this capability pre-empts the use of apostrophes for delimiting string literals. Attempts to pre-process programs utilitizing apostrophes for string literals will generate syntax errors of the form:

```
            missing apostrophe in character literal
```

An utility `ratfix' is available for quickly correcting such code.

Integer Constants:

Integer constants in bases other than decimal may be specified as n%dddd... where 'n' is a decimal number indicating the base and 'dddd...' are digits in that base. For bases > 10, letters are used for digits above 9. Examples include: 8%77 (=63), 16%2ff (=767), 2%0010011 (=19). The number is converted to the equivalent decimal value using multiplication; this may cause sign problems if the number has too many digits.

Lines and Continuation:

Input is free-format; that is, statements may appear anywhere on a line, and the end of the line is generally considered the end of the statement. However, lines ending in special characters such as comma, +, -, and * are assumed to be continued on the next line. An exception to this rule is within a condition; the line is assumed to be continued if the condition does not fit on one line. Explicit continuation is indicated by ending a line with an underline character (_). The underline character is not copied to the output file.

Comments:

Comments are preceded by '#' signs and may appear anywhere in the code.

Literal (unprocessed) Lines:

Lines can be passed through ratfor without being processed by putting a percent "%" as the first character on the line. The percent will be removed and the line shifted one position to the left, but otherwise will be output without change. Macro invocations, long names, etc., appearing in the line will not be processed.

Literal (unprocessed) Character Sequences:

Sequences of characters can be passed through the processor, thus avoiding processing, by surrounding then with the tokens %(...%). The surrounding %[()] tokens will be removed and the character sequence will be output without change. Macro invocations, long names, etc. appearing in the character sequence will NOT be processed.

Long Variable Name Processing:

An optional capability available in the pre-processor, which may be enabled by your local tools support individual, is the capability of converting long variable names (those consisting of more than 6 alpha-numerics, embedded underscores, or both) to 6 character ANSI-66 compliant variable names. If this option is available, and has been used in a pre-processing run, a sequence of FORTRAN comment statements are output at the end of the generated FORTRAN code, with the mapping of long names to generated names.

It should be noted that this mapping is not deterministic across separate compilations; as such, if ‘get_next_input’ is compiled and placed in a library, source invocations of ‘get_next_input’ would not map into the identical 6-character name. To permit users to preload the long name table with the names of external routines, the ‘linkage’ statement may be used:

```
                linkage long_name external_name
```

The pair of names is entered into the table of known long variable names, preventing any generated names for local long variables from colliding with the external name. The programmer must provide accurate information via this statement to permit access to routines with "long variable names" across compilations.

If long variable name processing has not been enabled for your

site, linkage is synonymous with define.

NOTE: since long variable name processing is optional, its use will generate code that is inherently non-portable to sites not desiring this capability. Users wishing to write portable code should avoid long variable names.

CHANGES

This ratfor preprocessor differs from the original (as released by Kernighan and Plauger) in the following ways:

The code has been rewritten and reorganized.

Hash tables have been added for increased efficiency in searching for macro definitions and Ratfor keywords.

The 'string' declaration has been included.

The define processor has been augmented to support macros with arguments.

Conditional preprocessing upon the definition (or lack therof) of a symbol has been included.

Many extraneous gotos have been avoided.

Blanks have been included in the output for increased readability.

Multi-level 'break' and 'next' statements have been included.

The Fortran 'DO' is allowed, as well as the ratfor one.

The capability of specifying integer constants in bases other than decimal has been added.

Underscores have been allowed in names.

The 'define' syntax has been expanded to include the form: define name value

The 'return(value)' feature has been added.

Quoted file names following 'include' statements have been added to allow for special characters in file names.

A method for allowing lines to pass through un-processed has

been added.

The 'switch' control statement has been included.

Continuation lines have been implemented.

Brackets have been allowed to replace braces (but NOT '$(' and '$)')

Character constants are now supported.

Groups of FORTRAN statements are permitted in the init and re-init clauses of the for statement.

A method for allowing character sequences to pass through un-processed has been added.

An 'undefine' command has been added to permit removal of symbol definitions.

Three types of literal character string processing are now possible. The default action permanently eliminates the usage of Hollerith constants in portable tools.

Long variable names processing can now be enabled as a site-dependent option.

FILES

A generalized definition file (e.g. 'ratdef') is automatically opened and read.

SEE ALSO

Kernighan and Plauger's "Software Tools"
Kernighan's "RATFOR - A Preprocessor for a Rational Fortran"
The Unix command rc in the Unix Manual
The tools 'incl' and 'macro'

DIAGNOSTICS

(The errors marked with asterisk '*' are fatal; all others are simply warning messages.)

* arg stack overflow
  The argument stack for the macro processor has been exceeded. The size of the stack is determined by the symbol ARGSIZE in the source definitions file.

o arith error
  An error occurred while evaluating the built-in macro, 'arith'.

* buffer overflow

One of the preprocessor's internal buffers overflowed, possibly, but not necessarily, because the string buffers were exceeded. The definition SBUFSIZE in the preprocessor symbols file determines the size of the string buffers.

* call stack overflow
  The call stack (used to store call frames) in the macro processor has been exceeded. The definition CALLSIZE in the source definition file determines the size of this stack.

* cannot make identifier unique
  All attempts to generate an unique short variable name for the long variable name being processed failed. This message will only be seen if the long variable name processing has been enabled.

o cannot open standard definitions file
  The special file containing general purpose ratfor definitions could not be opened, possibly because it did not exist or the user did not have access to the directory on which it resides.

o can't open include
  File to be included could not be located, the user did not have privilege to access it, or the file could not be opened due to some problem in the local primitives.

o conditional processing still active at EOF
  A sufficient number of "enddef" directives have not been encountered before detecting EOF on the input file.

* Conditionals nested too deeply
  The stack for nested conditionals has overflowed. The size of the stack is specified by the value of COND_STACK_DEPTH defined in the preprocessor symbols file.

* definition too long
  The number of characters in the name to be defined exceeded Ratfor's internal array size. The size is defined by the MAXTOK definition in the preprocessor symbols file.

o duplicate case label
  Two case labels with identical values were detected.

* EOF in string
  The macro processor detected an EOF in the current input file while evaluating a macro.

* evaluation stack overflow
  The evaluation stack for the macro processor has been exceeded. This stack's size is determined by the symbol EVALSIZE in the source definition file.

* for clause too long
  The internal buffer used to hold the clauses for the 'for' statement was exceeded. Size of this buffer is determined by the MAXFORSTK definition in the preprocessor symbols

```
        file.
    * getdef is confused
        There  were  horrendous  problems when attempting to access
        the definition table
    o illegal break
        Break did not occur  inside  a  valid  "while",  "for",  or
        "repeat" loop
    o illegal case or default
        A  "case" or "default" statement was detected which was not
        in the scope of a "switch" statement.
    o illegal case syntax
        The case label  was  not  of  the  correct  form.   It  may
        consist   of   comma-separated   constants   or  ranges  of
        constants.
    o illegal else
        Else clause probably did not follow an "if" clause
    * Illegal enddef encountered
        An "enddef" directive  was  encountered  while  conditional
        preprocessing was inactive.
    o illegal next
        "Next"  did  not  occur  inside  a valid "for", "while", or
        "repeat" loop
    o illegal range in case label
        A case label specifying a range  of  values  (of  the  form
        m-n) was detected in which m > n.
    o illegal right brace
        A right brace was found without a matching left brace
    o in entdef: no room for new definition
        There  is  insufficient  memory for macro definitions, etc.
        Increase the MEMSIZE definition in the preprocessor.
    o includes nested too deeply
        There is a limit  to  the  level  of  nesting  of  included
        files.   It  is dependent upon the maximum number of opened
        files  allowed  at  a  time,  and  is  set  by  the  NFILES
        definition in the preprocessor symbols file.
    o invalid case label
        The  upper  limit  of  a  case label specifying a range was
        non-numeric.
    * invalid conditional token
        The  token  given  as  the  argument  to  an   "ifdef"   or
        "ifnotdef" directive was not alpha-numeric.
    o invalid for clause
        The  "for"  clause did not contain a valid init, condition,
        and/or increment section
    o invalid string size
        The string format 'string name(size) "..."' was  used,  but
        the size was given improperly.
    * missing `(' in conditional
        The   first   non-blank   token  following  an  "ifdef"  or
        "ifnotdef" directive was NOT a left parenthesis.
```

* missing `)' in conditional
    An "ifdef" of "ifnotdef" directive was not properly terminated with a right parenthesis.
* missing `)' in define
    A define(...) was not properly terminated with a right parenthesis.
* missing `(' in undefine
    The first non-blank token following an "undefine" was NOT a left parenthesis.
* missing `)' in undefine
    An "undefine" directive was not properly terminated with a right parenthesis.
o missing apostrophe in character literal
    An apostrophe-delimited string NOT of the form 'c' or '@c' was encountered.
* missing colon in case or default label
    The list of case labels, or the default label were not followed by a colon.
* missing comma in define
    Definitions of the form 'define(name,defn)' must include the comma as a separator.
o missing function name
    There was an error in declaring a function
o missing left brace in switch statement
    The left brace indicating the start of the block of case labels for the "switch" statement was not encountered.
o missing left paren
    A parenthesis was expected, probably in an "if" statement, but not found
o missing literal quote
    The terminating "%)" to a literally quoted string was not found.
o missing parenthesis in condition
    A right parenthesis was expected, probably in an "if" statement, but not found
o missing quote
    A quoted string was not terminated by a quote
o missing right paren
    A right parenthesis was expected in a Fortran (as opposed to Ratfor) statement but not found
o missing string token
    No array name was given when declaring a string variable
* multiple defaults in switch statement
    More than one "default" statements were detected in the scope of a single "switch" statement.
o No room for generated variable name
    The table space used for generated long variable names has been exhausted. Increase the MEMSIZE definition in the preprocessor. This message cannot appear unless the long variable name processing has been enabled.

- o No room for linkage external name
  The table space used for generated external names has been exhausted. Increase the MEMSIZE definition in the preprocessor. This message cannot appear unless the long variable name processing has been enabled.
- * non-alphanumeric name
  Definitions may contain only alphanumeric characters and underscores.
- * stack overflow in parser
  Statements were nested at too deep a level. The stack depth is set by the MAXSTACK definition in the preprocessor symbols file.
- * switch table overflow
  More case labels were specified than the internal storage can handle. The size of the internal storage is determined by the value of MAXSWITCH defined in the preprocessor symbols file.
- o token too long
  A token (word) in the source code was too long to fit into one of Ratfor's internal arrays. The maximum size is set by the MAXTOK definition in the preprocessor symbols file.
- * too many characters pushed back
  The source code has illegally specified a Ratfor command, or has used a Ratfor keyword in an illegal manner, and the parser has attempted but failed to make sense out of it. The size of the push-back buffer is set by BUFSIZE in the preprocessor symbols file.
- o unbalanced parentheses
  Unbalanced parentheses detected in a Fortran (as opposed to Ratfor) statement
- o unexpected EOF
  An end-of-file was reached before all braces had been accounted for. This is usually caused by unmatched braces somewhere deep in the source code.
- o warning: possible label conflict
  This message is printed when the user has labeled a statement with a label in the 23000-23999 range. Ratfor statements are assigned in this range and a user-defined one may conflict with a Ratfor-generated one.
- * "file": cannot open
  Ratfor could not open an input file specified by the user on the command line.

AUTHORS

Original by B. Kernighan and P. J. Plauger, with rewrites and enhancements by David Hanson and friends (U. of Arizona), Joe Sventek and Debbie Scherrer (Lawrence Berkeley Laboratory), and Allen Akin (Georgia Institute of Technology).

BUGS/DEFICIENCIES

Missing parentheses or braces may cause erratic behavior. Eventually Ratfor should be taught to terminate parenthesis/brace checking at the end of each subroutine.

Although one bug was fixed which caused line numbers in error messages to be incorrect, they still aren't quite right. (newlines in macro text are difficult to handle properly). Use them only as a general area in which to look for errors.

Extraneous 'continue' statements are generated within Fortran 'do' statements. The 'next' statement does not work properly when used within Fortran 'do' statements.

There is no way to explicitly cause a statement to begin in column 6 (i.e. a Fortran continued statement), although implicit continuation is performed.

Ratfor is very slow, principally in the lexical analysis, character input, and macro processing routines (in that order). Attempts to speed it up should concentrate on the routines 'gtok', 'ngetch', and 'deftok'. An even better approach would be to re-work the lexical analyzer and parser completely.

NAME
    Ratp2 - Ratfor second pass processor

SYNOPSIS
```
ratp2 [file] ... >outfile
```

DESCRIPTION
    'ratp2' is the second pass of the new pre-processor. It's function is to re-order the output of the first pass to be ANSI-66 compliant. It's input is simply FORTRAN code, and all statements between successive END statements are re-ordered. If filename arguments are not provided, it reads from standard input.

SEE ALSO
    ratfor, the ratfor preprocessor, for descriptions of the language.

AUTHORS
    Phil Scherrer wrote ratp2.

BUGS/DEFICIENCIES

NAME
    Rc - RatFor compiler

SYNOPSIS
    rc [-cdfmorv] file ...

DESCRIPTION
    rc is the ratfor compiler. It accepts the following types of arguments:

1. Files whose names end in '.r' are assumed to be ratfor source programs; they are preprocessed into fortran and compiled. The preprocessed file for name.r is placed on name.f and the compiled object code appears on name.obj. The name.f file is removed unless -f is specified (see below).

2. The flags which affect the actions of the compiler are:

   -c suppress the loading phase, as does any preprocessing or compilation error

   -d do whatever is necessary to prepare the fortran files for the system debugger. In addition, pass the -d on to fc. The -d implies -f also.

   -f save fortran intermediate files; usually for debugging purposes

   -m passed on to fc and ld. Produce a load map of some sort.

   -o generates fortran listing for name.f on name.l

   -r ratfor only; don't compile fortran; implies -f and -c

   -v verbose option; prints additional information about the compilation process

3. Files whose names end in '.f' are assumed to be fortran source programs, and are compiled. Other arguments are assumed to be loader flags, or object files, typically created by an earlier rc or fc run. These files, together with the results of any compilations, are loaded to produce an executable process.

SEE ALSO
    ratfor, the ratfor preprocessor, for descriptions of the language and for a more general way of performing the

preprocessing.
fc, the fortran compiler
ld, the loader, for loader flags and process naming conventions

AUTHORS

Joe Sventek wrote the interface of rc to ratfor, fc, and ld.

BUGS/DEFICIENCIES

NAME
    Resume - resume a suspended process

SYNOPSIS
    resume processid [processid ...]

DESCRIPTION
    resume resumes a suspended process which has been suspended by the utility suspnd. The processid's are returned by the shell when a background process is spawned.

FILES
    none

SEE ALSO
    suspnd - suspend a running process
    sh - shell (command line interpreter)

DIAGNOSTICS
    If the process cannot be resumed, an error message will be displayed on the error output.

AUTHORS
    Joe Sventek (VAX)

BUGS/DEFICIENCIES

NAME

Rev - reverse lines

SYNOPSIS

rev [file] ...

DESCRIPTION

Rev copies the named files to the standard output, reversing the order of the characters in every line.

If no files are given, or the filename '-' is specified, rev reads from the standard input.

AUTHORS

David Hanson and friends (U. of Arizona)

DIAGNOSTICS

BUGS/DEFICIENCIES

NAME
    Rm - remove files

SYNOPSIS
    rm [-fiv] [file] ...

DESCRIPTION
    rm removes the files specified. If none are specified and standard input is not a terminal, ‘rm’ reads the names of the files to delete from the standard input. The options are:

    -v (verbose) display each file’s name as it is deleted

    -f (force) attempt deletion regardless of protection

    -i (interactive) prompt for confirmation before deleting unless the "-f" option is in effect.

    If a file is protected from delete access, you are asked if you want to try anyway. If you respond with a "y", rm will try to unprotect the file and then delete it.

FILES

SEE ALSO
    The Unix command ’rm’

DIAGNOSTICS
    A message is printed if the file could not be removed.

AUTHORS
    Joe Sventek (DEC machines); Debbie Scherrer (CDC machines) The "-f" and "-i" options were added by Dave Martin.

BUGS/DEFICIENCIES

NAME
    Ruler - display ruler on terminal screen

SYNOPSIS
    ruler [n]

DESCRIPTION
    ruler displays a ruler on the terminal. This is especially useful when using field or other utilities which require knowledge of the column positions of portions of the screen. The optional numeric argument indicates how many columns to format in the ruler.

FILES

SEE ALSO
    field - utility for field manipulation
    sort - file sorter

DIAGNOSTICS

AUTHORS
    Dave Martin

BUGS/DEFICIENCIES

NAME

Sched - a way to repetitively invoke a command

SYNOPSIS

sched [-r<repetitions>] [-t<seconds>] "shell command"

DESCRIPTION

sched causes the command typed in quotes to be repetitively invoked. The defaults are to invoke the command once, and to wait 1 second before each invocation. This utility is quite nice for statistics gathering, since sched may be run in the background, with the diagnostic output being appended to some log file. For example:

```
% sched -r144 -t600 "who | lcnt >>usrcnt"
```

would generate a log of the number of users on the system for one day, running at 10-minute intervals. The resulting list of numbers could then be fed to a suitable analysis or plotting program.

FILES

SEE ALSO

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME

Sedit - stream editor

SYNOPSIS

```
sedit [-n]  [[-e] command] ... [-f commandfile] ... [file] ...
```

DESCRIPTION

sedit copies the input files (default is standard input) to the standard output, performing one or more editing commands (see 'ed') on each line.

The -n flag indicates that only lines that are explicitly printed by 'p' commands are to be copied to the standard output. Double copies of some lines will be output if the 'p' command is used without specifying the -n flag.

The -e flag indicates that the next argument is a sedit command.

The -f flag indicates that the next argument is the name of a file in which sedit commands appear one per line.

The -e and -f arguments may be intermixed in any order. The order of command execution is the order in which commands are read.

If no -e or -f flags are given, the first argument is used as an sedit command. When the first argument not in the scope of a flag is encountered, it and all succeeding arguments are taken as input files. If no files are given, or if the name "-" is specified, the standard input is read.

Sedit commands have the general form

```
line1 [, line2] [!] command arguments
```

A line number (line1 or line2) is either a decimal number that refers to a specific input line (input lines are counted cumulatively across files), a "$" that refers to the last line of input, or a /pattern/ where pattern is a regular expression (as in 'ed'). Line number 0 may be used to specify commands that should be executed before any input is read.

A command with no line numbers is applied to every line of input. A command with one line number is applied to every line of input that matches the line number. A command with two line numbers is applied to every line of input beginning with the first line that matches line1 through the next line that matches line2. Thereafter, the process is repeated, looking again for a line that matches line1.

A command is negated by placing the '!' character after the line numbers and before the command character. This has the effect of executing the command on all of the lines except the ones specified.

There is no notion of '.' and no relative addressing. No expressions in addresses are allowed. There are no backward pattern searches with '\'. A 'p' at the end of a command only works with the 's' command.

If an 'a', 'i', 'c', or 'r' command is successfully executed, the text is inserted into the standard output whether or not the line on which the match was made is later deleted or not. Text inserted in the output stream by these commands is not scanned for any pattern matches, nor are any sedit commands applied to it, nor will it effect the input line numbering.

Sedit accepts the following commands. Each command may be used with 0, 1, or 2 line numbers. Any of the commands may appear on the 'sedit' command line except the a, c, and i commands. They can only be used in command files.

a
<text>
.
Append. The <text> is placed on the output after each selected line.

c
<text>
.
Change. The selected lines are deleted and <text> is placed on the output in their place.

d
Delete. The selected lines are deleted.

i
<text>
.
Insert. The <text> is placed on the output before each selected line.

p
Print. The selected lines are printed on the standard output.

q
Quit. The current line is output (unless the -n option is specified) and no further processing is done.

r file
   Read file. The contents of "file" are placed on the output after each selected line exactly as if the contents were given as <text> in an a command.

s/pat/new/gp
   Substitute. The leftmost occurrences of pat in the selected lines are changed to new. If g is specified, all occurrences are changed. If p is specified, the resulting line is printed. The search string 'pat' is a regular expression as defined for 'ed'. The replacement string 'new' also uses the same conventions as 'ed' for search string replacement (&, and $1...$9). Subsequent sedit commands will only match the resulting lines.

w file
   Write file. The selected lines are appended to "file". Files mentioned in w commands are created before processing begins. The limit on the number of w commands depends on the number of files that can be opened at the same time.

=
   Print line number. The current line number is printed on the output as a line.

Sedit can accomodate commands (including <text> arguments), totaling approximately 5000 characters (20,000 if LARGE_ADDRESS_SPACE is defined).

SEE ALSO
ed, change, tr

DIAGNOSTICS
In addition to the usual error messages resulting from file access failure, sedit issues the following messages preceded by the offending command line.

bad line numbers
   indicates that the line number expressions are invalid.

invalid command
   indicates that the command preceeding the message is illegal. This message is issued for a, i, or c commands if they appear in command string scripts.

too many commands
   indicates exhaustion of space to hold commands. The size of the command buffer is determined by the MAXBUF definition in the source code.

AUTHORS
    Layne Cannon (Battelle Northwest Labs)
    Chris Fraser (U. of Arizona)

BUGS/DEFICIENCIES

NAME
    Send - send a message to another user's terminal

SYNOPSIS
    send {user | -user | term}

DESCRIPTION
    Send copies lines from your terminal to that of another user. When first called, it sends the message

```
     [message from <your_name> on <your_terminal> hh:mm:ss]
```

    All lines you type will then be transmitted to the other user's terminal until you enter a ^Z. The message

```
     [end of message from <your_name> hh:mm:ss]
```

    is then sent.

    You may specify either a username or a particular terminal (i.e. tta0) to receive the message. If you specify a username and that user is logged in on more than one terminal, you are asked to pick one of the terminals to receive the message. If -username is specified then all of the terminals that the user is logged in on will receive the message.

FILES
    A scratch file generated with seed ``who''.

IMPLEMENTATION
    Send spawns ``who'' to map users to their terminals, and then calls the VMS SYS$BRDCST system service to send the messages.

SEE ALSO
    The UNIX command "write"

DIAGNOSTICS
    ? Can't write to ``username''.

    ? Can't spawn ``who''.

    ? Can't read scratch file.

AUTHORS
    Dave Martin (Hughes Aircraft) with modifications by Mike Kimura.

BUGS/DEFICIENCIES

NAME
    Sepfor - Split FORTRAN programs into multiple files

SYNOPSIS
    sepfor [-v] file ...

DESCRIPTION
    Sepfor is useful for cracking large FORTRAN programs into separate files. Each subroutine or function is placed in a file of the same name. Names are stripped of any ``$'' and ``_'' characters they may contain. The main program (which is assumed to precede the subroutines in the source file) is named ``main<n>'' where <n> is the number of the file argument. In most cases there is only one file specified and the main program is thus named ``main1''.

    If the ``-v'' (verbose) option is specifed, Sepfor echoes the name of each routine on STDOUT as it is processed.

EXAMPLES
```
    sepfor -v spice.for
```

FILES
    none

IMPLEMENTATION
    Sepfor decides it has found a subroutine when it finds the keyword ``subroutine'' as the first word on a line. It decides it has found a function when it finds the keyword ``function'' as the the second OR third word on a line. The name is taken to be the first word following the keyword. Sepfor decides it has found the end of a module when it discovers the keyword ``end'' at the beginning of a line and it does NOT find the keyword ``do'' or ``if'' immediately thereafter.

AUTHORS
    Dave Martin (Hughes Aircraft)

BUGS/DEFICIENCIES
    Sepfor does not recognize ENDDO or ENDIF; you must separate the keywords with a blank.

NAME

Sh - shell (command line interpreter)

SYNOPSIS

sh [-cdnvx] [name [arguments]].

DESCRIPTION

Sh is a command line interpreter: it reads lines typed by you and interprets them as requests to execute other programs.

o COMMANDS

In simplest form, a command line consists of the command name followed by arguments to the command, all separated by spaces:

```
command arg1 arg2 ... argn
```

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 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' can't be found in the current directory or through its pathname, the shell searches your 'home/tools' directory, the site-specific tools directory, and finally the general tools directory. If the file still has not been found, and the '-d' switch has not been specified, the shell passes the entire command line to the local operating system's command line interpreter (DCL for VMS). An example of a simple command is:

```
sort list
```

which would sort the contents of file 'list', printing the output at your terminal.

Some characters on the command line have special meanings to the shell (these are discussed below). The character '@' may be included anywhere in the command line to cause the following character to lose any special meaning it may have to the shell (to be 'escaped'). Sequences of characters enclosed in double (") or single (') quotes are also taken literally.

o STANDARD I/O

Shell programs in general have three standard files open : 'input', 'output', and 'error output'. All three are assigned to your terminal unless redirected by the special arguments '<', '>', '?', '>>', '??', (and sometimes '-').

An argument of the form '<name' causes the file 'name' to be used as the standard input file of the associated command.

An argument of the form '>name' causes file 'name' to be used as the standard output.

An argument of the form '?name' causes the file 'name' to be used as the standard error output.

Arguments of the form '>>name' or '??name' cause program output to be appended to 'name' for standard output or error output respectively. If 'name' does not exist, it will be created.

Most tools have the capability to read their input from a series of files. In this case, the list of files overrides reading from standard input. However, many of the tools allow you to read from both a list of files and from input by specifying the filename '-' for standard input. For example:

```
format file1 - file2
```

would read its input from 'file1', then from the standard input, then from 'file2'.

o FILTERS AND PIPES

The output from one command may be directed to the input of another. A sequence of commands separated by vertical bars ('|') or carets ('^') causes the shell to arrange that the standard output of each command be delivered to the standard input of the next command in sequence. Thus in the command line:

```
sort list | uniq | crt
```

'Sort' sorts the contents of file 'list'; its output is passed to 'uniq', which strips out duplicate lines. The output from 'uniq' is then input to 'crt', which prepares the lines for viewing on your crt terminal.

The vertical bar is called a 'pipe'. Programs such as 'sort', 'uniq', and 'crt', which copy standard input to standard output (making some changes along the way) are called 'filters'.

o COMMAND SEPARATORS

Commands need not be on different lines; instead they may be separated by semicolons:

```
                    ar t file; ed
```

The above command will first list the contents of the archived file 'file', then enter the editor.

The shell also allows commands to be grouped together with parentheses, where the group can then be used as a filter. For example:

```
              (date; cat chocolate) | comm vanilla
```

writes first the date and then the file 'chocolate' to standard output, which is then read as input by 'comm'. This tool compares the results with existing file 'vanilla' to see which lines the two files have in common.

o MULTITASKING

On many systems the shell also allows processes to be executed in the background. 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 instance:

```
                    ratfor ambrose >george &
```

preprocesses the file 'ambrose', putting the output on 'george'. No matter how long the compilation takes, the shell returns immediately. 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. Parentheses and pipes are also allowed (within the same background process).

o SCRIPT FILES

The shell itself is a command, and may be called recursively, either implicitly or explicitly. This is primarily useful for executing files containing lines of shell commands. For instance, suppose you had a file named 'nbrcount.sh' which looked like this:

```
echo "Counting strings of digits"
tr <program 0-9 9 | tr !9 | ccnt
```

These commands count all the digit strings in 'program'. You could have the shell execute the commands by typing:

```
sh nbrcount.sh
```

The shell will also execute script files implicitly. For example, giving the command:

```
nbrcount
```

would cause the shell to notice that the file 'nbrcount.sh' contained text rather than executable code. The shell would then execute itself again, using 'nbrcount.sh' as its input.

Arguments may also be passed to script files. In script files, character sequences of the form '$n', where n is a digit between 1 and 9, are replaced by the nth argument to the invocation of the shell. For instance, suppose the file 'private.sh' contained the following commands:

```
cat $1 $2 $3 | crypt key >$4
ar u loveletters $4
```

Then, executing the command:

```
private Dan John Harold fair
```

would merge the files 'Dan', 'John', and 'Harold', encrypt them, and store them away in an archive under the name 'fair'.

Script files may be used as filters in pipelines just like regular commands.

Script files sometimes require in-line data to be available to them. A special input redirection notation '<<' is used to achieve this effect. For example, the editor normally takes its commands from the standard input. However, within a shell procedure commands could be embedded this way:

```
ed file <<!
{ editing requests }
!
```

The lines between '<<!' and '!' are called a 'here' document; they are read by the shell and made available as the standard input. The character '!' is arbitrary, the document being

terminated by a line that consists of whatever character followed the '<<'.

You may establish scripts for the shell to execute when you 'login' to a shell by creating a script file named 'login.sh' in your home/tools directory.

o SEARCH PATH

When the shell receives a command to execute, such as

```
% tool
```

it looks for 'tool' in the following places, in the following order:

1) 'tool.sh' in the current working directory
2) 'tool.xxx' in the current working directory, where 'xxx' is to be replaced by the appropriate extension for an image file on your system.
3) ~/tool.sh or ~/tools/tool.sh
4) ~/tool.xxx or ~/tools/tool.xxx
5) ~usr/tool.sh
6) ~usr/tool.xxx
7) ~bin/tool.sh
8) ~bin/tool.xxx

The search stops whenever one of these files is found; the type of the file (ASCII | BINARY) is then determined. If the type is BINARY, then a sub-process running that image file is spawned; otherwise, a sub-process running the shell is spawned, with that shell reading the located file as its input commands. If the entire search path is exhausted without success, the command is handed to the native command interpreter for execution, unless the '-d' option has been selected.

o SHELL FLAGS

The shell accepts several special arguments when it is invoked. The argument '-v' asks the shell to print each line of a script file as it is read as input. For instance,

```
          sh -v private Jasmine Irma Jennifer twostars
```

would print each line of the script file 'private' as soon as it is read by the shell.

The argument '-x' is similar to the -v above except that commands are printed right before they are executed. These commands will be printed in the actual format the system expects when attempting to execute the program.

The argument '-n' suppresses execution of the command entirely.

The argument '-c' causes the remaining arguments to be executed as a shell command.

The argument '-d' inhibits the shell from 'dropping through' to the native command line interpreter when a command can't be found.

o INTERNAL COMMANDS

Several commands are actually executed by the shell itself. As such, they cannot have the standard I/O units redirected. The syntax and semantics of these commands are:

* von

  Enables the -v flag above.

* voff

  Disables the -v flag.

* xon

  Enables the -x flag above.

* xoff

  Disables the -x flag.

* cd [directory]

  Changes the current working directory (CWD) to the specified directory. If the single argument is omitted, the CWD is changed to the last directory visited in this way. If the change of the CWD fails, an error message is displayed and the CWD is left unchanged.

* ho[me]

  Change the current working directory to the user's home directory. The same result can be achieved via 'cd ~/'.

* logout

  Causes the shell to stop reading the current input file. This is equivalent to an EndofFile on the current input file.

* # [args]

  This command is a comment. This permits script files to be commented for future enlightenment. A blank character MUST separate the '#' from the comment strings.

* path

  Display the search path in current use.

* alias
  alias name
  alias name value

  The first form lists the values of all known aliases. The second form lists the value of the alias 'name'. The third form creates an alias 'name' having 'value'. 'value' is simply taken to be the remainder of the command, with parameter substitution being performed on the words. See the section below on aliases and parameters for more information.

* unalias name

  Destroy the alias 'name'. See the section below on aliases and parameters for more information.

* param
  param name
  param name value

  The first form lists the values of all known parameters. The second form lists the value of the parameter 'name'. The third form creates a parameter 'name' having 'value'. 'value' is simply taken to be the remainder of the command, with parameter substitution being performed on the words. See the section below on aliases and parameters for more information.

* unparam name

  Destroys the parameter 'name'. See the section below on aliases and parameters for more information.

* ask name[prompt[default-value]]

  Prompts the user on the Standard Input unit for the value of the parameter 'name'. If the prompt string is not specified, or is null (""), the string "name? " will be used. If the user responds with a bare carriage-return, the parameter will assume the default value, if specified, or will not be defined.

* source file

  The current input unit is stacked, and the shell input is taken from 'file'. 'source' commands nest to a maximum depth of 2. Upon detection of an EndofFile on 'file', input is resumed from the previous input file.
  ***** NOTE: source commands must appear alone on a line, or dire consequences will result! *****

o ALIASES AND PARAMETERS

Often it is convenient to store frequently used strings in variables for recall with a small number of keystrokes. Aliases and parameters exist to provide such a facility, differing only in the way that they are used.

When the shell has finished parsing your command is and in the process of preparing to execute it, the first token in the command line (the verb) is looked up in the table of aliases. If it is found, then the verb is replaced by the value of the alias; independent of the replacement of the verb, the command line is then executed. For instance, if you with to invoke the editor with a personalized prompt, the following alias

```
alias e ed "-pWas gibt? "
```

causes the following transformation to take place

```
e file  ====>  ed "-pWas gibt? " file
```

The user must explicitly ask for a parameter to be expanded. We have already seen examples of the use of parameters, when referencing the positional arguments to scripts as $1, $2, ..., $9. For example, suppose that a particular directory on another machine has a set of files with cooking recipes. A parameter can be used to permit easy reference to the directory

```
param cook /0de/db0/frenchchf
```

Then commands of the form

```
ls $cook; cat $cook/quiche.man
```

will permit you to list the contents of the directory and display one of the recipes.

Parameters are expanded inside of quoted strings when they are delimited by a quote character ("), but are not expanded when delimited by an apostrophe ('). In addition to the positional parameters $1, $2, ..., $9, two shorthand parameters are available for causing all positional parameters to be displayed:

```
$@    results in "$1" "$2" ...
$*    results in "$1 $2 ..."
```

o INTERRUPTS

There are often occasions when you may wish to interrupt the execution of a process initiated by the shell. This may be achieved by typing the interrupt character at the terminal. Typing the interrupt character will cause the process to be terminated, and the shell will prompt you for your next command. A complete list of system-specific special terminal characters may be had by typing the command 'tty' to the shell. 'Tty' is a system-dependent tool which displays on standard output all of the special terminal characters interpreted by the local system. For example, the interrupt character for the VAX is ^C (control C). If the following two commands are typed to the shell:

```
sort mybigfile
^C
```

then the sorter would be aborted.

o TERMINATION

The shell may be terminated by typing an EndOfFile ('^Z') as a command.

FILES

SEE ALSO
The Unix command sh.
The Bell system Technical Journal, vol. 57, no. 6, part 2, July-Aug 1978.

DIAGNOSTICS

The error message 'syntax error' appears whenever a command line cannot be understood.

AUTHORS

Dennis Hall, Joe Sventek, Debbie Scherrer, Dave Martin.

BUGS/DEFICIENCIES

If you want to escape a shell special character that appears as the first character of an argument, you must escape it with quotes rather than an '@' sign.

NAME

Sleep - cause process to suspend itself for a period of time

SYNOPSIS

sleep seconds

DESCRIPTION

sleep causes the process to suspend itself for the indicated number of seconds. This facility is generally useful when sending formatted output to a high-quality terminal, and you need time to change the paper from the time you invoke the command until it starts printing on the good paper.

FILES

SEE ALSO

sched - a way to repetitively invoke a command

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME

Sort - sort and/or merge text files

SYNOPSIS

```
sort [-bdfimr] [+ofile] [+sn] [file] ...
```

DESCRIPTION

Sort sorts lines of all the named files together and writes the result on the standard output. The name '-' means the standard input. The standard input is also used if no input file names are given. Thus sort may be used as a filter.

The sort key is an entire line. Default ordering is alphabetic by characters as they are represented in ASCII format. The ordering is affected by the following flags, one or more of which may appear.

-b Leading blanks are not included in keys.

-d 'Dictionary' order: only letters, digits and blanks are significant in comparisons.

-f Fold all letters to a single case.

-i Ignore all nonprinting nonblank characters.

-m Merge only, the input files are already sorted.

-r Reverse the sense of the sort

+o Cause final output to be placed on 'file'. This permits one of the input files to be the output file. This switch is necessary since using the redirection '>file' will cause 'file' to be unreadable when 'sort' is generating the initial runs.

+sn Sort according to the subfield starting on column n

FILES

A series of scratch files are generated and subsequently deleted. Presently the files are named "STn" where "n" is a sequence number.

SEE ALSO

The Unix command "sort" in the Unix User's Manual.

DIAGNOSTICS

A message is printed if a file cannot be located.

AUTHORS

Original design from Kernighan and Plauger's "Software Tools", with modifications by Debbie Scherrer. The external merge phase of sort was completely rewritten by Joe Sventek.

BUGS/DEFICIENCIES

The merge phase is performed with a polyphase merge/sort algorithm, which requires an end-of-run delimiter on the scratch files. The one chosen is a bare ^D(ASCII code 4) on a line. If this is in conflict with your data files, the symbol CTRLD in sortsym should be redefined and sort built again.

Eventually all the Unix "sort" flags should be implemented. These include:

```
     sort [-mubdfinrtx] [+pos] [-pos] [-o file] [file] ...
```

The additional flags are:

n An initial numeric string, consisting of optional minus sign, digits and optionally included decimal point, is sorted by arithmetic value.

tx Tab character between fields is x.

+pos -pos Selected parts of the line, specified by +pos and -pos, may be used as sort keys. Pos has the form m.n optionally followed by one or more of the flags bdfinr, where m specifies a number of fields to skip, n a number of characters to skip further into the next field, and the flags specify a special ordering rule for the key. A missing .n is taken to be 0. +pos denotes the beginning of the key; -pos denotes the first position after the key (end of line by default). Later keys are compared only when all earlier keys compare equal. Note: The first field of a line is numbered zero.

When no tab character has been specified, a field consists of nonblanks and any preceding blanks. Under the -b flag, leading blanks are excluded from a field. When a tab character has been specified, fields are strings separated by tab characters.

Lines that otherwise compare equal are ordered with all bytes significant.

-o The next argument is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs, except under the merge flag -m. {Note--it is not clear why this flag is needed.]

−u Suppress all but one in each set of contiguous equal lines. Ignored bytes and bytes outside keys do not participate in this comparison.

NAME

Spell - find spelling errors

SYNOPSIS

```
spell [-ddictname] [file] ...
```

DESCRIPTION

Spell copies the named files (or standard input if none are specified) to standard output while looking up each word in a dictionary. If any spelling errors are found in a particular line, an additional line will be printed immediately following the line with asterisks (*) beneath the offending words.

If the -d switch is used, 'spell' will use the files 'dictname' and 'dictname'dx for the dictionary and index.

FILES

dict - a dictionary file
dictdx - the index generated by isam for the dictionary

SEE ALSO

isam - generate an index for pseudo-indexed-sequential access
ospell - the script pipeline suggested in K&P for spelling errors

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

This is a skeleton spelling error detector. It is expected that various modifications to flesh it out will be performed for local use.

NAME

Split - split a file into pieces

SYNOPSIS

```
split [-n] [file [name] ]
```

DESCRIPTION

Split reads 'file' and writes it in n-line pieces (default 1000), as many as necessary, onto a set of output files. The name of the output file is 'name' with 'aa' appended, and so on lexicographically. If no output name is given, 'x' is default.

If no input file is given, or if - is given in its stead, then the standard input file is used.

FILES

SEE ALSO

The Unix command 'split'

DIAGNOSTICS

A message is printed if the input file could not be opened.

AUTHORS

Debbie Scherrer

BUGS/DEFICIENCIES

NAME
    Suspnd - suspend a running process

SYNOPSIS
    suspnd processid [processid ...]

DESCRIPTION
    suspnd suspends running processes specified by the processid's in the command line. The processid's are those returned by the shell when it spawns a background process.

FILES
    none

SEE ALSO
    sh - shell (command line interpreter)
    resume - resume a suspended process

DIAGNOSTICS
    if the process cannot be suspended, an error message is displayed on error output.

AUTHORS
    Joe Sventek (VAX)

BUGS/DEFICIENCIES

NAME

Tail - print last lines of a file

SYNOPSIS

```
tail [-n] [file] ...
```

DESCRIPTION

Tail prints the last "n" lines of the indicated file. If 'n' is omitted, the last 23 lines are printed.

If "file" is omitted or is "-", tail reads the standard input.

SEE ALSO

split

AUTHORS

David Hanson and friends (U. of Arizona)

BUGS/DEFICIENCIES

An internal buffer of MAXBUF characters is kept. If the value of "n" would require buffering more characters than the buffer can hold, tail prints the last MAXBUF characters of the file. In this case, the first line of output may not be an entire line. MAXBUF is a definition in the source code which may be adjusted.

NAME
    Tee - copy input to standard output and named files

SYNOPSIS
    tee [file] ...

DESCRIPTION
    Tee copies the standard input to the standard output and makes copies in the named files.

FILES

SEE ALSO
    The tool 'cat'; the tool 'crt'; the Unix command 'tee'

DIAGNOSTICS
    A message is printed if the input file cannot be opened.

AUTHORS
    Debbie Scherrer

BUGS/DEFICIENCIES

NAME

Timer - time execution of a process

SYNOPSIS

```
timer [-v] "command [arguments]"
```

DESCRIPTION

timer spawns a subprocess performing the requested command, and displays the CPU time and wall time which elapsed during the execution of the command on the standard output. The -v flag causes timer to display other system-dependent quantities concerning the subprocess performing the requested command. The command specified is searched for using the same search path as the shell.

FILES

none

SEE ALSO

The UNIX programmer's manual, time(I)
sh - shell (command line interpreter)

DIAGNOSTICS

<command> is an invalid image or script file name
: The requested command could not be found in the searched directories.

Error in spawning <command>
: The requested image or script file was located, but the process to perform the command could not be spawned.

AUTHORS

Joe Sventek(VAX)

BUGS/DEFICIENCIES

NAME

Tr - transliterate characters

SYNOPSIS

```
tr from [to]
```

DESCRIPTION

tr copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in 'from' are mapped into the corresponding characters of 'to'. Ranges of characters may be specified by separating the extremes by a dash. For example, a-z stands for the string of characters whose ascii codes run from character a through character z.

If the number of characters in 'from' is the same as in 'to', a one to one corresponding translation will be performed on all occurrences of the characters in 'from'. If the number of characters in 'from' is more than in 'to', the implication is that the last character in the 'to' string is to be replicated as often as necessary to make a string as long as the 'from' string, and that this replicated character should be collapsed into only one. If the 'to' string is missing or empty, "TR" will take this condition as a request to delete all occurrences of characters in the 'from' string.

"TR" differs from the tool "CH" since it deals only with single characters or ranges of characters, while "CH" deals with character strings. For example tr xy yx would change all x's into y's and all y's into x's, whereas ch xy yx change all the patterns "xy" into "yx".

One of the most common functions of "TR" is to translate upper case letters to lower case, and vice versa. Thus,

```
tr A-Z a-z
```

would map all upper case letters to lower case. Users of systems which cannot pass both upper and lower case characters on a command line should remember to include the appropriate escape flags.

FILES

none

SEE ALSO

Tools "find" and "ch".
The "Software Tools" book, p.51-61.
The "UNIX Programmer's Manual", p. TR(I).

DIAGNOSTICS

"usage: tr from [to]."
: The command line passed to transit is in error.

"from: too large."
: The string for "from" is too large. Current limit is 100 characters including E0S.

"to: too large."
: The string for "to" is too large. Current limit is 100 characters including EOS.

AUTHORS

Original code from Kernighan and Plaugers's "Software Tools", with modifications by Debbie Scherrer.

BUGS/DEFICIENCIES

NAME

Tsort - topologically sort symbols

SYNOPSIS

```
tsort [file] ...
```

DESCRIPTION

tsort topologically sorts the symbols in the named files. If no files are specified, or the filename '-' is given, tsort reads the standard input.

A symbol is considered any string of characters delimited by blanks or tabs.

Each line of the input is assumed to be of the form

```
a b c ...
```

which states that a precedes b, a precedes c, and so on. Note that there is nothing implied about the ordering of b and c. A line consisting of a single symbol simply "declares" that symbol without specifying any ordering relations about it. The output is a topologically sorted list of symbols, one per line.

For example, suppose you have trouble getting up in the morning because you can't quite remember what actions have to be performed in which order. However, you do know that the first action in the following list precedes all others on the line:

```
set_alarm   turn_off_alarm
wake_up    get_out_of_bed    turn_off_alarm
set_alarm     wake_up
```

Using tsort to sort the above list would produce the following set of actions for getting out of bed:

```
set_alarm
wake_up
turn_off_alarm
get_out_of_bed
```

DIAGNOSTICS

circular
: The input specifies a graph that contains at least one cycle.

out of storage
: The input is too large. The size of tsort's buffer is determined by the MAXBUF definition in the source code.

SEE ALSO
    sort

AUTHORS
    David Hanson and friends (U. of Arizona)

BUGS/DEFICIENCIES

NAME

Ttt - 3-dimensional tic tac toe

SYNOPSIS

ttt

DESCRIPTION

TTT is a 3-dimensional tic tac toe game played against the computer. The program will explain the rules.

SEE ALSO

The UNIX ``ttt'' program.

AUTHORS

Original Basic version by Joseph Roehrig.
Converted to C by Dave Conroy.
Converted to RatFor by Dave Martin.

BUGS/DEFICIENCIES

NAME

Txtrpl - perform generalized text replacement

SYNOPSIS

txtrpl patfile ...

DESCRIPTION

'txtrpl' provides a general way to perform text replacement (NOT regular expressions) without embedding the (text,replacement text) pairs in the source file. After loading the (text,replacement text) pairs from the named pattern files in the command line, 'txtrpl' reads words from standard input, looks each word up in a lookup table, and either writes out the replacement text on standard output or the word, depending upon whether it was found in the table or not. Only a single lookup is done. Words consist of letters, digits and underline ('_') characters, starting with a letter.

'txtrpl's selection of candidate words for replacement is dependent upon ratfor program syntax, in that words inside of comments, quoted strings and character constants are not eligible for replacement. This fact can be exploited to generate source listings of ratfor code with boldfaced keywords by executing the following commands:

```
alist file | txtrpl ~bin/fmtpf
```

The resulting output file can be piped into 'os' or 'lpr' for final disposition to a print device.

The form of the pattern files is quite simple; each (text,replacement text) pair occupies a line. Leading blanks on the line are ignored, the token to be scanned for is the first word found, any intevening blanks are ignored, and the replacement text is everything else up to the end of line. In the regular language expression of the tools, each line is of the form

```
%<BLANK>*[A-Za-z][A-Za-z0-9_]*<BLANK><BLANK>*??*$
```

where <BLANK> represents a blank character. Case is important in the comparisons.

FILES

SEE ALSO

macro - macro processor
ed - text editor for description of regular expressions
xch - extended change utility

DIAGNOSTICS

AUTHORS
    Joe Sventek

BUGS/DEFICIENCIES

NAME
    Ul - convert backspaces into multiple lines for "terminals"

SYNOPSIS
    ul [file] ...

DESCRIPTION
    ul (underline) converts lines with BACKSPACE-UNDERLINE pairs into two lines; one with the text and one with only BLANK and UNDERLINE characters. These two lines are output separated by a CR with no associated LF. This approach works with printing terminals and some line printers, most notably Printronix and Trilog.

    If no files are given, or the filename '-' appears, input is taken from the standard input.

FILES

SEE ALSO
    lpr - queue file to line printer
    os - process overstrikes for "printers"

DIAGNOSTICS
    A message is printed if an input file cannot be opened; further processing is terminated.

AUTHORS
    Paul Johnstone (Hughes Aircraft)

BUGS/DEFICIENCIES

NAME
    Uniq - strip adjacent repeated lines from a file

SYNOPSIS
    uniq [-c] [file] ...

DESCRIPTION
    uniq reads the input file(s), comparing adjacent lines. Second and succeeding copies of repeated lines are removed; the remainder is written to standard output.

    If the '-c' flag is given, each line is preceded by a count of the number of occurrences of that line.

FILES

SEE ALSO
    The tool 'comm'; the Unix command 'uniq'

DIAGNOSTICS
    A message is printed if an input file cannot be opened and processing is terminated.

AUTHORS
    Originally from Kernighan and Plauger's 'Software Tools', with modifications by Debbie Scherrer.

BUGS/DEFICIENCIES

NAME

Unrot - unrotate lines rotated by kwic

SYNOPSIS

```
unrot [-n] [file] ...
```

DESCRIPTION

unrot processes the rotated output of 'kwic' to generate a keyword-in-context index.

The -n flag may be used to specify the width of the output lines. The default is 80.

If no input files are given, or the filename '-' appears, lines will be read from standard input.

FILES

SEE ALSO

kwic; sort

DIAGNOSTICS

A message is printed if an input file cannot be opened; further processing is terminated.

AUTHORS

Original from Kernighan and Plauger's 'Software Tools', with modifications by Debbie Scherrer.

BUGS/DEFICIENCIES

NAME

Wc - count lines, words, and characters in files

SYNOPSIS

```
wc [-lwc] [file] ...
```

DESCRIPTION

wc prints the number of lines, words, and characters in the named files. The filename "-" specifies the standard input. A total is also printed. A "word" is any sequence of characters delimited by white space.

The options -l, -w, and -c specify, respectively, that only the line, word, or character count be printed. For example,

```
wc -lc foo
```

prints the number of lines and characters in "foo".

If no files are given, wc reads its standard input and the total count is suppressed.

FILES

DIAGNOSTICS

name: can't open
    Printed when an input file can't be opened; processing ceases

AUTHORS

David Hanson and friends (U. of Arizona)

BUGS/DEFICIENCIES

NAME

Wcnt - (character) word count

SYNOPSIS

```
wcnt [file] ...
```

DESCRIPTION

wcnt counts (character) words in the named files, or in the standard input if no name appears. A word is a string of characters delimited by spaces, tabs, or newlines.

wcnt could also be implemented as a shell script file:

```
               tr ' @t@n' '@n' | tr '!@n' | ccnt
```

FILES

SEE ALSO

lcnt; ccnt; the Unix command 'wc'

DIAGNOSTICS

A message is printed if an input file could not be opened; further processing is terminated.

AUTHORS

Original from Kernighan and Plauger's 'Software Tools', with modifications by Debbie Scherrer.

BUGS/DEFICIENCIES

NAME

Whereis - locate file in tree based on partial pathname

SYNOPSIS

whereis pat [anchor]

DESCRIPTION

'whereis' recursively scans a directory tree looking for the regular expression given as the first argument. If no 'anchor' argument is supplied, 'whereis' starts looking in the current directory and throughout the directory tree descending from the current directory. If 'anchor' is specified, the search starts at that directory. Valid patterns are the same as those for 'ls'. The output from 'whereis' on standard output are fully resolved pathnames, complete with device information. It should be noted that a valid 'anchor' argument is "/", which indicates to 'whereis' that it should start looking in the "root" directory of the current disk, or "/dba0" to force it to start looking in the "root" directory for dba0:.

FILES

none

SEE ALSO

ls - directory lister
find - find pattern, for regular expression syntax

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME

Who - show who is on the system

SYNOPSIS

```
who [-htv] [am i]
```

DESCRIPTION

who lists the name and terminal number for each current system user. The following switches affect the format of the listing:

- -h Generate a line of column headers above the display.
- -t Print the total number of users at bottom of display.
- -v Generate a verbose display, with system dependent information constituting the verbose portion.

FILES

SEE ALSO

The Unix command 'who'

DIAGNOSTICS

AUTHORS

Joe Sventek (DEC machines); Sheldon Furst (CDC machines)

BUGS/DEFICIENCIES

NAME
    Xch - extended change utility

SYNOPSIS
    xch [-gpat] [-v] patfile ...

DESCRIPTION
    'xch' permits several global changes to be performed during one pass over the input data. During initialization, 'xch' compiles the "/pat/sub/" lines found in the one (or more) pattern files specified in the arguments list. Then, standard input is read, and the equivalent of

```
    % ch "pat" "sub"
```

    is performed on each line.

    Normally, the substitutions are attempted on each input line. If the '-gpat' option is selected, then the substitutions are attempted on only those lines which match 'pat'. When the number of substitutions are large, this can substantially speed up the process.

    If the '-v' flag is specified, for each line in which a substitution has occurred, the line number, followed by the old and new lines are displayed on error output.

    The format of the pattern files is quite simple: each "/pat/sub/" pair occupies a single line, with the first character of the line assumed to be the delimeter character. The complete regular expression syntax is supported, such that the lines in the pattern files are exactly equivalent to the 'ed' command with "s" prepended and "g" appended to the line.

FILES

SEE ALSO
    ch - change regular expressions
    find - find regular expressions
    xfind - extended find utility
    ed - text editor

DIAGNOSTICS

AUTHORS
    Joe Sventek

BUGS/DEFICIENCIES

NAME

Xfind - entended find utility

SYNOPSIS

xfind patfile ...

DESCRIPTION

'xfind' permits one to search for more than 10 expressions in one pass of the standard input file. During initialization, 'xfind' compiles the patterns found in the one (or more) pattern files specified in the argument list. Then, standard input is read, and each input line which matches any one of the patterns is output on standard output.

The format of the pattern fils is quite simple: each line is taken to represent a single pattern. The complete regular expression syntax is supported.

FILES

SEE ALSO

find - find regular expressions
xch - extended change utility

DIAGNOSTICS

AUTHORS

Joe Sventek

BUGS/DEFICIENCIES

NAME

Xref - make a cross reference of symbols

SYNOPSIS

```
xref [-b<bias>] [-f] [file] ...
```

DESCRIPTION

xref produces a cross-reference list of the symbols in each of the named files on the standard output. Each symbol is listed followed by the numbers of the lines in which it appears. If no files are given, or the file "-" is specified, xref reads the standard input.

A symbol is defined as a string of letters, digits, underlines, or periods that begins with a letter. Symbols exceeding an internal limit are truncated. This limit is determined by the MAXTOK definition in the source code, and is currently set to 15.

Normally, xref treats upper- and lower-case letters as different characters. The -f option causes all letters to be folded to lower-case.

Normally, the line numbers specified in the symbol table are relative to the current file being processed. Specification of the '-b' flag causes '<bias>' to be added to each line number.

DIAGNOSTICS

out of storage

The file contains too many symbols or references to fit within the current limitations of xref. The size of the buffer is determined by the MAXBUF definition in the source code.

SEE ALSO

axref - cross reference generator for archives

AUTHORS

David Hanson and friends (U. of Arizona)

BUGS/DEFICIENCIES

There should be a means of suppressing "junk" symbols such as "the", "a", etc.

```
 ____            _   _               ____
/ ___|  ___  ___| |_(_) ___  _ __   |___ \
\___ \ / _ \/ __| __| |/ _ \| '_ \    __) |  _____
 ___) |  __/ (__| |_| | (_) | | | |  / __/  |_____|
|____/ \___|\___|\__|_|\___/|_| |_| |_____|

 ____            _                    ____      _ _
/ ___| _   _ ___| |_ ___ _ __ ___    / ___|__ _| | |___
\___ \| | | / __| __/ _ \ '_ ` _ \  | |   / _` | | / __|
 ___) | |_| \__ \ ||  __/ | | | | | | |__| (_| | | \__ \
|____/ \__, |___/\__\___|_| |_| |_|  \____\__,_|_|_|___/
       |___/
```

NAME
    Intro - introduction to software tools primitives

A (Not So) Primitive Document

Joseph Sventek
Computer Science & Applied Mathematics
Lawrence Berkeley Laboratory
Berkeley, CA 94720

A complete writeup of the syntax and semantics of the primitive functions upon which the LBL Software Tools Virtual Operating System is based.

Basic assumptions of the tools concerning data types:

'character' is a signed integer data type of at least eight-bit accuracy. The internal representation of characters within the programs is the 7-bit ASCII code, with EOS being traditionally defined as 0 (NULL). This leaves 128 negative values for special flags such as EOF, ERR etc. Msg assumes that it can use 200(8) (NULL with the high bit set) to pad NEWLINES in raw terminal I/O.

Integers and characters are freely assigned to each other. Since the above assumption is made, normal FORTRAN compilers should perform this without any side effects.

The data type 'linepointer' should be defined as an intrinsic FORTRAN data type large enough to hold the address of a record in a file. No arithmetic or assignments are done on linepointers, with all activities on these entities embodied in the routines 'note', 'seek', 'ptreq', 'ptrcpy', 'ptrtoc' and 'ctoptr'. It is also assumed that a defined symbol NULLPOINTER exists and is different from any possible valid linepointer value.

Basic assumptions of the ratfor runtime system

I/O redirection flags are interpreted (and removed from the argument list) before the tool's main routine is called. In order to be able to pass arguments to the utility which start with the special characters '<', '>', or '?', quoted string arguments are not scanned when determining io redirection. The primitives, as currently defined, only handle character files, with the exception that 'gettyp' expects to be able to determine whether a file is BINARY or ASCII. It is expected that the primitives will be extended in the near future to handle ASCII, LOCAL and BINARY files unambiguously.

'getarg(0)' should return the name by which the process was invoked. This is useful if your system supports UNIX-style links, thus allowing an image to act differently depending upon which alias was used for its invocation. None of the tools currently use this capability.

The first routine to be executed is named

```
subroutine main
```

It is assumed that all I/O redirection and command line fetching have been completed before 'main' is called. Upon completion, 'main' simply returns, which causes a return to the Tools runtime system. In order to implement this feature, 'ld' is set up to extract a module from the library which is the FORTRAN main program, and usually consists of the following lines:

```
call initst
call main
call endst(OK)
end
```

If such library extraction is not possible on a system, the above four line routine will have to be added to each tool.

Basic assumptions concerning RAW terminal I/O

The routine 'stmode' is supplied to permit the use of RAW terminal I/O. In actuality, three modes are defined, RAW, RARE and COOKED. COOKED io implies that the system applies its own semantics concerning the control characters emitted by the terminal, and performs the echo of the characters for the user. RARE and RAW assume that all reads are done a character at a time, with no echo. RARE io assumes that there are certain control characters which the operating system will not relinquish its control of. These probably include XON, XOFF, terminal interrupt characters, etc. RARE is the mode of io used by all of the tools currently using RAW io, since they usually only wish to apply their own semantics to the actual printing characters. RAW io is handy if one wishes to write network control programs over asynchronous lines in ratfor (Don't laugh, it's being done!). With RAW io, it is assumed that the terminal driver is totally bypassed. On many systems, this capability requires enormous privilege and other funky resources, so none of the commonly available tools use it.

NAME

Amove - move (rename) file1 to file2

SYNOPSIS

```
integer function amove(name1, name2)

character name1(ARB), name2(ARB)
```

DESCRIPTION

'amove' moves the contents of the file specified by 'name1' to the file specified by 'name2'. It is essentially a renaming of the file.

Both file names are character strings representing pathnames or filenames in whatever format is expected by the local operating system. The names are passed as character arrays terminated with an EOS character.

The files need not be opened by (connected to) the running program to be renamed.

The function value returned is OK is successful or ERR if not.

IMPLEMENTATION

'amove' could be easily implemented by opening the first file, creating the second, copying the first to the second, and then removing the first file. Alternatively, if possible, it could be implemented with a native system call to rename the file.

SEE ALSO

remove(2)

DIAGNOSTICS

If the rename fails for any reason, ERR is returned. 'name1' is removed only if the rename succeeds.

NAME

Assign - open a file on the specified unit

SYNOPSIS

```
integer function assign(name, fd, access)

character name(FILENAMESIZE)
filedes fd
integer access
```

DESCRIPTION

'assign' is the equivalent of 'open' or 'create' on a particular ratfor I/O unit. If a file is currently open on 'fd', 'assign' closes it first. If 'access' has the value of READ, 'assign' then performs an 'open' on the specified unit. If 'access' has the value of WRITE, READWRITE or APPEND, 'assign' performs a 'create' on the specified unit. The function value returned is either the value of 'fd' if successful, or ERR.

IMPLEMENTATION

There has been much debate whether 'assign' should still be in the primitive set. The only tool which relies upon it is 'sort', since is does some fairly complex file manipulation during the external merge phase.

SEE ALSO

open(2), create(2), close(2)

DIAGNOSTICS

If the file could not be opened or created for any reason, the value ERR is returned. In this case, the previous file associated with 'fd' remains closed.

NAME

Brdcst - broadcast message to one or all terminals

SYNOPSIS

```
integer function brdcst(msg, dev)

character msg(ARB), dev(ARB)

return(OK/ERR)
```

DESCRIPTION

'brdcst' broadcasts the message in 'msg' to the terminal specified by the 'dev' argument. If 'dev' is the string "all", the message is broadcast to all logged in terminals on the system.

IMPLEMENTATION

'brdcst' is heavily dependent upon whether the operating system supports such a notion. In addition, some systems support broadcasts only for very privileged users. This routine is only used by 'sndmsg' and 'mail' to notify users of mail delivery, and can safely be implemented as a stub.

SEE ALSO

trmlst(2)

DIAGNOSTICS

Returns ERR if the message cannot be broadcast.

NAME
    Chmod - change protection mode on file

SYNOPSIS
```
    integer function chmod( name, mode)

    character name(ARB)
    integer mode

    return(OK/ERR)
```

DESCRIPTION
    'chmod' attempts to change the protection on the file 'name' to the value specified in 'mode'. It returns the value OK/ERR reflecting the degree of success in the operation.

IMPLEMENTATION
    The only current use of 'chmod' is in the 'rm' tool using the '-f' flag. In that situation, 'mode' is passed as an integer of all ones. Before 'chmod' can become generally useful, some system-independent way of specifying the protection on a file needs to be devised. It is totally permissible to implement this as a stub always returning the value ERR.

SEE ALSO
    rm(1)

DIAGNOSTICS
    Return a value of ERR if the mode change could not be performed.

NAME
    Closdr - close an opened directory

SYNOPSIS
    subroutine closdr(fd)

    filedes fd

DESCRIPTION
    'closdr' closes the directory that is currently opened and associated with the internal descriptor 'fd', which was returned by the 'opendr' function.

IMPLEMENTATION

SEE ALSO
    opendr(2)

DIAGNOSTICS
    If 'fd' is an invalid descriptor, or if no opened directory is currently associated with 'fd', 'closdr' returns with no error message.

NAME

Close - close (detach) a file

SYNOPSIS

```
subroutine close(fd)

filedes fd
```

DESCRIPTION

'close' closes the connection between a file and the running program. Any write buffers are flushed and the file is rewound.

'fd' is an internal file descriptor as returned from an 'open' or 'create' call.

IMPLEMENTATION

'close' breaks the connection between the program and a file accessed via 'open' or 'create'. If necessary, the file's write buffer is flushed and the end of the file is marked so that subsequent reads will find an EOF. If a file has been opened multiple times (that is, more than one internal descriptor has been assigned to a file), care is taken that multiple closes will not damage the file.

SEE ALSO

open(2), create(2)

DIAGNOSTICS

If the file descriptor is in error, the routine simply returns.

NAME

Create - create a new file (or overwrite an existing one)

SYNOPSIS

```
filedes function create( name, access)

character name(ARB)
integer access
```

DESCRIPTION

'create' creates a new file from within a running program and connects the external name of the file to an internal identifier which is then usable in subsequent subroutine calls. If the file already exists, the old version will be overwritten. In this case, the file should be truncated immediately by 'create'.

'name' is a character string representing a pathname or filename in whatever format is used by the local operating system. It is passed as a character array terminated by an EOS character.

'access' is a integer descriptor for the type of access desired - WRITE, READWRITE or APPEND.

The value returned is a "filedes" internal descriptor to be used in subsequent I/O calls on this file.

IMPLEMENTATION

'create' is similar to 'open' except that 'create' generates a new file if it does not already exist, whereas 'open' returns an error on such occasions.

SEE ALSO

open(2), close(2)

DIAGNOSTICS

The function returns ERR if the file could not be created or if there are already too many files open.

NAME

Ctoptr - convert character string into linepointer

SYNOPSIS

```
subroutine ctoptr(buf, i, ptr)

character buf(ARB)
integer i
linepointer ptr
```

DESCRIPTION

'ctoptr' converts the characters starting at location 'buf(i)' into a linepointer value and stores the value in the variable 'ptr'. The value of 'i' is incremented to point to the next available location in 'buf'.

IMPLEMENTATION

SEE ALSO

ptreq(2), ptrcpy(2), note(2), seek(2), ptrtoc(2)

DIAGNOSTICS

none

NAME

Cwdir - change current working directory

SYNOPSIS

```
integer function cwdir(name)

character name(FILENAMESIZE)
```

DESCRIPTION

'cwdir' changes the current working directory to that specified by 'name'. 'name' is a character string representing a pathname or whatever format is expected by the local operating system. 'name' is passed as a character array terminated by an EOS character.

The return value is either OK or ERR depending upon the success of the operation. If the operation fails, the current working directory should be restored to the previous value.

IMPLEMENTATION

SEE ALSO

gwdir(2)

DIAGNOSTICS

A value of ERR is returned if the operation is unsuccessful.

NAME

Delarg - mask the existence of specified command line argument

SYNOPSIS

```
subroutine delarg(n)

integer n
```

DESCRIPTION

'delarg' masks the existence of the command line argument number 'n' so that subsequent calls to 'getarg' do not see it.

IMPLEMENTATION

'delarg' works in conjunction with 'getarg'. It generally re-orders indices to an array holding the command line arguments fetched by 'makarg'. 'delarg' is currently only used by the shell.

SEE ALSO

getarg(2), initst(2)

DIAGNOSTICS

If argument 'n' does not exist, 'delarg' simply returns.

NAME
Enbint - enable trapping of terminal interrupts

SYNOPSIS
subroutine enbint

DESCRIPTION
'enbint' is used by the shell to trap interrupt characters typed by the user at the terminal. 'enbint' assumes that there will be a routine named 'intsrv' which will be called whenever a terminal interrupt is typed. The canonical semantics of 'intsrv' is to kill all sub-processes of the current process and return. This generally results in the return of error notifications to 'spawn', which returns the error to the shell, after which the shell prompts for another command.

IMPLEMENTATION
'enbint' has been implemented on only three machines, and is not very well defined. In all of the implementations to date, 'enbint' checks to make sure that the caller is the top shell in the process tree associated with the user. This prevents 'enbint' from being generally called from other programs. It is hoped that a firmer specification for this routine will be available in the near future.

If this routine is difficult to implement, it may be left as a stub.

SEE ALSO
intsrv(2)

DIAGNOSTICS
If the enabling of the interrupt cannot be performed, the current implementations simply return.

NAME
    Endst - perform system-dependent cleanup and terminate ratfor program

SYNOPSIS

```
    subroutine endst(status)

    integer status
```

DESCRIPTION
    'endst' is normally implicitly called when the 'main' subroutine executes a return. 'endst' closes all open files, performs any necessary system-dependent cleanup and terminates the program's execution.

    If it is possible, endst should communicate the termination status (OK/ERR/CHILD_ABORTED) to the outside world.

    'endst' is also called by 'error' to terminate the program.

IMPLEMENTATION

SEE ALSO
    close(2), initst(2)

DIAGNOSTICS
    none

NAME

Filnfo - determine filename and access on open unit

SYNOPSIS

```
integer function filnfo( fd, file, access)

filedes fd
integer access
character file(FILENAMESIZE)
```

DESCRIPTION

'filnfo' returns the name and access of the file open on 'fd' to the user. If the unit is open, 'filnfo' returns OK as its value, otherwise it returns ERR.

IMPLEMENTATION

SEE ALSO

DIAGNOSTICS

If the file specified by 'fd' is not open, a value of ERR is returned.

NAME

Gdraux - get auxiliary information about a file

SYNOPSIS

```
subroutine gdraux( fd, file, aux, date, fmtstr)

character file(FILENAMESIZE), aux(MAXLINE), date(TCOLWIDTH)
character fmtstr(ARB)
filedes fd
```

DESCRIPTION

'gdraux' retrieves auxiliary information on a particular file in a directory. 'fd' is the directory descriptor returned from an 'opendr' call and 'file' is a filename returned from a 'gdrprm' call. The auxiliary information is returned in the character array 'aux', while 'date' receives a "sortable" date string of size (TCOLWIDTH-1) which can be used to sort files by significant date.

The information placed into 'aux' is dependent upon the format string passed in 'fmtstr'. The format string specifies the output information as follows:

b	size of file in blocks (normally 512 characters)
c	size of file in characters
m	modification date and time (dd-mmm-yy hh:mm:ss)
n	filename
o	file owner's username
p	protection codes
t	file type (asc|bin|dir)

The 'b', 'c', 'n' and 'o' options accept an integer prefix which specifies the field width to be used.

IMPLEMENTATION

This is admittedly a stop-gap measure until a more useful and penetrating primitive is devised to permit the retrieval of extra information about a file. The only utility which uses 'gdraux' currently is 'ls', the directory lister. The sortable date field can be anything that the primitive implementor desires, but it is strongly suggested that it be a sortable version of whatever significant date the operating system keeps on the file, so that the "-t" flag in 'ls' performs up to specification.

EXAMPLES

The verbose option of ‘ls’ uses the format string "17n p m 6b o".

SEE ALSO

opendr(2), gdrprm(2)

DIAGNOSTICS

If the auxiliary information cannot be obtained for a particular file, a message to that effect is returned in ‘aux’, and ‘date’ is given a value such that it will sort out first when sorting by date.

NAME
    Gdrprm - get next filename from directory

SYNOPSIS
```
    integer function gdrprm( fd, file)

    character file(FILENAMESIZE)
    filedes fd
```

DESCRIPTION
    'gdrprm' retrieves the next sequential filename from the open directory associated with 'fd' and places it in the character array 'file' as an EOS-terminated string. If there is an error reading the directory or no more filenames are contained in the directory, a value of EOF is returned; otherwise, OK is returned. The filenames are retrieved sequentially, with no particular order (alphabetic, by date, etc.) guaranteed.

IMPLEMENTATION
    If there are lots of noise characters (version numbers, null extensions, etc.), these are often stripped from the filename before it is returned.

SEE ALSO
    gdraux(2), opendr(2)

DIAGNOSTICS
    A value of EOF is returned whenever there are no more directory entries or an error reading the directory is detected.

NAME

Getarg - get command line arguments

SYNOPSIS

```
integer function getarg( n, array, maxsiz)

character array(maxsiz)
integer n, maxsiz
```

DESCRIPTION

'getarg' gets command arguments from the command line or control card and copies the 'n'th command line argument into the character array 'array', terminating it with an EOS character. 'maxsiz' is passed as the maximum number of characters 'array' is prepared to deal with (including the EOS character); 'getarg' truncates the argument if necessary to fit into the space provided. The number or characters in the argument (not including the EOS character) is returned in the functional call. If there are less than 'n' arguments, EOF is returned. Calling 'getarg' with 'n' having the value of 0 should result in the return of the name by which the image was invoked.

IMPLEMENTATION

The implementation of 'getarg' may be quite different on different operating systems. Some systems allow only upper case (or lower case) on the command line; they may limit size; they may not even provide access at all without considerable contortions.

When implementing 'getarg', the designer should keep in mind that a 'delarg' will also be needed. One possible design would be to create a routine 'makarg', which would pick up the arguments from the system, convert them to ascii strings, handle any upper-lower case escape conventions, and store them in an array. 'getarg' could then access this array, stripping off any quoted strings surrounding the arguments, and passing them along to the user. 'delarg' could also access this array when removing reference to arguments.

If it is absolutely impossible to pick up command line arguments from the system, 'getarg' could be taught to prompt the user for them.

When the shell is implemented, 'getarg' (or perhaps 'makarg') may have to be altered to read arguments as passed from the shell.

SEE ALSO

initst(2), delarg(2)

DIAGNOSTICS
    none

NAME

Getch - read character from file

SYNOPSIS

```
character function getch( c, fd)

character c
filedes fd
```

DESCRIPTION

'getch' reads the next character from the file specified by 'fd'. The character is returned in ASCII format both as the functional return and in the parameter 'c'. If the end of a line has been encountered, NEWLINE is returned. If the end of the file has been encountered, EOF is returned.

If the unit 'fd' is a RAW or RARE terminal unit, then getch actually gets the next character from the terminal WITH NO ECHO.

IMPLEMENTATION

Interspersed calls to 'getch' and 'getlin' should interleave properly. A common implementation is to have 'getlin' make repeated calls to 'getch'.

If the input file is not ASCII, characters are mapped into their ASCII equivalent.

SEE ALSO

getlin(2), putch(2), putlin(2), stmode(2)

DIAGNOSTICS

If an error occurs during the reading of the file, the value ERR is returned.

NAME
    Getdir - get directory string for known Tools directory

SYNOPSIS
```
    subroutine getdir( key, dtype, name)

    character name(FILENAMESIZE)
    integer key, dtype
```

DESCRIPTION
    'getdir' returns the directory string for any of the known Tools directories. The directory string is returned as a character array terminated by an EOS character. The format of 'name' is determined by the value of 'dtype', with LOCAL generating a string in local format, and PATH causes a pathname directory string to be returned. The valid values of 'key' and their corresponding directories are:

BINDIRECTORY	~bin/
USRDIRECTORY	~usr/
TMPDIRECTORY	~tmp/
LPRDIRECTORY	~lpr/
MSGDIRECTORY	~msg/
MANDIRECTORY	~man/
SRCDIRECTORY	~src/
INCDIRECTORY	~inc/
LIBDIRECTORY	~lib/

    If an invalid key is specified, a null string is returned.

IMPLEMENTATION

SEE ALSO

DIAGNOSTICS
    If an invalid key is specified, a null string is returned.

NAME

Getlin - get next line from file

SYNOPSIS

```
integer function getlin( line, fd)

character line(MAXLINE)
filedes fd
```

DESCRIPTION

'getlin' copies the next line from the file specified by the internal identifier 'fd' into the character array 'line'. Characters are copied until a NEWLINE character is found or until MAXLINE-1 characters have been copied. The characters are returned with the character array terminated by an EOS character.

'getlin' returns EOF when it encounters an end-of-file, otherwise it returns the line length (including NEWLINE, excluding EOS).

Interspersed calls to 'getlin' and 'getch' are permitted and should work properly.

IMPLEMENTATION

If the external representation of characters is not ASCII, the characters are mapped into their ASCII equivalents.

'getlin' assumes a maximum size (MAXLINE) of the array 'line'. If the input line exceeds the limit, only the first "limit-1" characters are returned, with the remainder of the line either being ignored or returned on the next 'getlin' call.

A common implementation is to have 'getlin' call getch until a NEWLINE character is found (or the buffer size is exceeded or EOF is reached).

If the underlying disk structure is record oriented (as opposed to stream oriented), it may be more efficient to have 'getlin' get the next record in the same way that 'getch' does, to avoid the overhead of repeated calls to 'getch'.

Use of 'getlin' on RAW terminal units is of questionable utility, since the repeated 'getch' calls perform a READ WITH NO ECHO, and would only terminate when the user types a CTRL/J (LINEFEED) character. All utilities which use RAW I/O have their own line gathering routines.

SEE ALSO

getch(2), putch(2), putlin(2), stmode(2)

DIAGNOSTICS
    none

NAME
    Getnow - determine current date and time

SYNOPSIS
```
    subroutine getnow (now)

    integer now (7)
```

DESCRIPTION
    'Getnow' is used to query the operating system for the current date and time. The requested information is returned in a seven-word integer array, where:

```
         word 1 contains the year (e.g. 1980);
         word 2 contains the month (e.g. 9);
         word 3 contains the day (e.g. 25);
         word 4 contains the hour (e.g. 13);
         word 5 contains the minute (e.g. 39);
         word 6 contains the second (e.g. 14);
         word 7 contains the millisecond (e.g. 397).
```

    The information returned by 'getnow' may be used as-is or further useful processing may be done by 'fmtdat' or 'wkday'.

IMPLEMENTATION
    Operating systems generally have some mechanism for picking up the current date and time. If yours has one, use it.

    Getnow is not critical to the implementation of the tools and can be left as a stub if the operating system cannot supply the needed information.

ARGUMENTS MODIFIED
    now

BUGS/DEFICIENCIES
    Some systems cannot obtain all the time information described. Array elements that cannot be filled default to zero.

SEE ALSO
    fmtdat(3), wkday(3), date(1)

NAME
    Gettyp - get type of file

SYNOPSIS
```
    integer function gettyp( fd, type)

    filedes fd
    integer type
```

DESCRIPTION
    'gettyp' determines whether the file opened on unit 'fd' is ascii characters (ASCII), local characters (LOCAL, if different from ASCII) or binary(BINARY). The type is returned as the value of the function and as the value of the parameter 'type'. If the file is empty or new, ASCII is returned.

    'fd' is the file identifier returned from an 'open' or 'create' call.

IMPLEMENTATION
    When a file is opened (via a call to 'open' or 'create'), an internal flag is usually set which specifies the file type. 'gettyp' then simply reads the flag. The file type may have been determined by locating system information about the file or by actually reading part of it and making a reasonable guess.

    'gettyp' is called by the archiver to store a file's type in the archiver header. The shell also uses 'gettyp' to determine whether a command verb given by the user represents a script file or an image file. If the verb corresponds to a character file, the shell spawns itself with the file as input. If 'gettyp' cannot be implemented on a particular system, a stub returning BINARY should be placed in the library, which will force the user to execute script files in the following manner:

```
    % sh <script ...
```

SEE ALSO
    create(2), open(2)

DIAGNOSTICS
    ERR is returned if the file descriptor is incorrect.

NAME
    Gtmode - determine mode of ratfor unit

SYNOPSIS
```
    integer function gtmode(fd)

    filedes fd
```

DESCRIPTION
    'gtmode' determines the mode of io on the unit 'fd', returning one of the values RAW, RARE or COOKED. If the unit is not currently opened, the value ERR is returned.

IMPLEMENTATION

SEE ALSO
    stmode(2)

DIAGNOSTICS
    If 'fd' is not currently open, a value of ERR is returned.

NAME

Gtzone - get time zone of requestor

SYNOPSIS

```
subroutine gtzone(buf)

character buf(4)
```

DESCRIPTION

'gtzone' returns to the requestor the 3-character mnemonic for the time zone, terminated by an EOS character.

IMPLEMENTATION

A typical way of implementing this routine is to simply strcpy a string into the buffer. This routine may return a null string.

SEE ALSO

DIAGNOSTICS

NAME

Gwdir - get current working directory

SYNOPSIS

```
subroutine gwdir( name, dtype)

character name(FILENAMESIZE)
integer dtype
```

DESCRIPTION

'gwdir' returns the current working directory string in 'name'. If 'dtype' has the value LOCAL, the directory string is returned in the form desired by the local operating system. If 'dtype' has the value of PATH, the directory string is returned in pathname format. The directory string is returned as a character array terminated by an EOS character.

IMPLEMENTATION

SEE ALSO

cwdir(2)

DIAGNOSTICS

none

NAME

Homdir - return the home directory for this process

SYNOPSIS

```
subroutine homdir(home, dtype)

character home(FILENAMESIZE)
integer dtype
```

DESCRIPTION

'homdir' returns the home directory string for the current process. If 'dtype' has the value LOCAL, the directory string is returned in the form native to the local operating system; a value of PATH causes it to be returned in pathname format. It is returned as an EOS terminated string. If this information cannot be determined, a stub which returns an EOS in home(1) will suffice.

IMPLEMENTATION

SEE ALSO

tooldr(3)

DIAGNOSTICS

none

NAME

Initst - initialize ratfor runtime environment

SYNOPSIS

subroutine initst

DESCRIPTION

Normally, 'initst' is implicitly called before the main subroutine of the user's program is called. 'initst' opens STDIN, STDOUT and ERROUT, performing any redirections specified in the command line and masking those redirections from the current process. The remainder of the command line arguments are prepared for retrieval via 'getarg', and any other system-dependent initialization is performed.

IMPLEMENTATION

The standard I/O units are generally opened in the order ERROUT, STDIN and STDOUT. If an error occurs during the opening of ERROUT, some system-dependent method of reporting the error will need to be used, whereas if an error occurs while opening STDIN or STDOUT, ERROUT can be used to report it. The fetching of command line arguments from the operating system is in the domain of 'initst', as well as any initializations of common data areas needed by the other primitive functions.

SEE ALSO

endst(2), getarg(2), delarg(2)

DIAGNOSTICS

If 'initst' cannot function for some reason, the program should abort with a diagnostic message.

NAME
Intsrv - interrupt service routine for tty interrupts

SYNOPSIS
subroutine intsrv

DESCRIPTION
'intsrv' is the routine called whenever a terminal interrupt character has been typed after 'enbint' has been called to enable the trapping of these interrupts. The canonical semantics of 'intsrv' is to kill all sub-processes of the current process. Other functions may be embedded in 'intsrv'.

IMPLEMENTATION
'enbint' and 'intsrv' are not very well defined. It is hoped that their definitions will become firmer in the near future.

If 'enbint' has been implemented as a stub, then 'intsrv' may also be implemented as such.

SEE ALSO
enbint(2)

DIAGNOSTICS
If during the course of its duties, 'intsrv' encounters an error, it should notify the user in some way. This may be tricky, due to the asynchronous nature of its work.

NAME
    Isatty - determine if file is an interactive device

SYNOPSIS
```
    integer function isatty(fd)

    filedes fd
```

DESCRIPTION
    'isatty' returns the value YES if the file specified by 'fd' is an interactive device, otherwise it returns NO. 'fd' is a file identifier returned by a call to 'open' or 'create'.

IMPLEMENTATION
    When a file is opened, a flag is usually set indicating what device the file is associated with. 'isatty' usually just reads this flag.

    'isatty' is used by several tools ('ls', 'ar' and 'users') to determine whether to format their output in columns or not. It may also be used to determine whether to prompt for input or not.

SEE ALSO
    open(2), create(2)

DIAGNOSTICS
    NO is returned if 'fd' is in error.

NAME

Loccom - locate command along specified search path

SYNOPSIS

```
integer function loccom( in, spath, suffix, out)

character in(FILENAMESIZE), out(FILENAMESIZE), spath(ARB)
character suffix(ARB)
```

DESCRIPTION

'loccom' searches for the command passed as an EOS-terminated character array in 'in' along the search path specified by 'spath', returning the fully-qualified file specification in the character array 'out'. For each element of the search path, all suffixes passed in 'suffix' are appended to 'in' and an open at READ access is attempted. The type of the file thus found (ASCII or BINARY) is returned as the value of the function. If the command cannot be found, the value ERR should be returned and the string 'in' copied to 'out'.

The structure of 'spath' and 'suffix' is:

```
string_1@estring_2@e...string_N@e@n
```

where '@e' represents an EOS character and '@n' represents a NEWLINE character. A null directory name indicates searching the current working directory.

IMPLEMENTATION

SEE ALSO

spawn(2)

DIAGNOSTICS

If the command cannot be found, a value of ERR is returned and the command string is returned in 'out'.

NAME

Mailid - return username

SYNOPSIS

```
subroutine mailid(user)

character user(USERSIZE)
```

DESCRIPTION

'mailid' returns the name of the user of the current process as an EOS terminated string. This name is then used by the mail system.

IMPLEMENTATION

Most operating systems permit the user to determine some unique identifier of the person (or account) on whose behalf the current process is running. The third field of each entry in the mail system's database file is dedicated to such an identifier, so that mailid could be implemented by determining the identifier, opening ~msg/address and reading records until an entry with that identifier is found, and returning the username (field one of the record) in the array 'user'. 'mailid' is essential for the correct working of the mail system.

SEE ALSO

homdir(2)

DIAGNOSTICS

If the record could not be found in the database, some nonsense username should be returned in 'user'.

NAME
    Mklocl - convert string to fully qualified local file specification

SYNOPSIS
```
    call mklocl( in, out)

    character in(FILENAMESIZE), out(FILENAMESIZE)
```

DESCRIPTION
    'mklocl' converts the input filename 'in', which may be in pathname format or a partial local file specification, to a fully qualified local file specification.

IMPLEMENTATION
    Initially, 'mklocl' could be a stub, simply 'scopy'ing 'in' to 'out'. 'mklocl' is used by the tools 'fc' and 'ld'.

SEE ALSO
    mkpath(2)

DIAGNOSTICS
    none

NAME
    Mkpath - convert string to fully resolved path name

SYNOPSIS

```
    call mkpath(in, out)

    character in(FILENAMESIZE), out(FILENAMESIZE)
```

DESCRIPTION
    'mkpath' converts the input filename 'in', which may be in pathname format or a partial local file specification, to a fully qualified pathname.

IMPLEMENTATION
    Initially, 'mkpath' could be a stub, simply 'scopy'ing 'in' to 'out'. 'mkpath' is used by the tools 'alist' and 'ls'.

SEE ALSO
    mklocl(2)

DIAGNOSTICS
    None

NAME
Note - determine current file position

SYNOPSIS
```
stat = note (offset, fd)

linepointer offset
filedes fd
integer stat returned as OK/ERR
```

DESCRIPTION
Note determines the current value of a file's read/write pointer. The argument "offset" is a linepointer that will receive the information. Offset is maintained untouched by the user and passed to "seek" when desiring to return to that particular location in the file.

Note is usually used as the file is being written, picking up the pointer to the end of the file before each record is inserted there.

On text files (e.g. those created by calls to putch, putlin), note is guaranteed to work at line boundaries only. However, it should work anywhere on a file created by calls to writef.

IMPLEMENTATION
Note is compatible with whatever implementation is chosen for seek and the opening of files at READWRITE access.

Offset is a linepointer in which is stored a character count, word address, block and record address, or whatever is appropriate for the local operating system. Note should be taught to return BEGINNING_OF_FILE and END_OF_FILE where appropriate.

In the editor, note is called to locate the end of file for subsequent writes.

SEE ALSO
seek(2), readf(2), writef(2)

DIAGNOSTICS
None

NAME
    Open - open an existing file

SYNOPSIS
```
    filedes function open( name, access)

    character name(ARB)
    integer access
```

DESCRIPTION
    'open' attaches an existing file to a running program and associates the external file name with an internal identifier for use in other I/O routines. Several opens of a file at READ access are permitted.

    'name' is a character string representing a pathname or filename in whatever format is used by the local operating system. It is passed as a character array terminated with an EOS character.

    'access' is a descriptor for the type of access desired - READ, WRITE, READWRITE or APPEND.

    The returned value of the function is a "filedes" descriptor to be used in other I/O routines when referring to this file.

    The file is positioned at the beginning, unless APPEND access is specified, in which case the file is prepared for extension.

IMPLEMENTATION
    'open' connects the file to the running program and performs those manipulations necessary to allow reading and/or writing to the file. An internal descriptor is assigned to the file and subsequently used when calling other primitives such as close, getch, putch, getlin, and putlin.

    'open' may have to set up an internal I/O buffer for the file. It may also have to do an initial read to determine the file type (ASCII or BINARY). Information about the file's type and teletype characteristics (YES or NO) is generally maintained. This information is then made available to the user via the 'gettyp' and 'isatty' functions.

    'open' is generally taught to read characters of ASCII type as well as LOCAL character type (if different from ASCII). Translation of characters from LOCAL to ASCII is done in getch/getlin and vice versa for putch/putlin.

    There is generally a limit to the maximum number of files open at any one time. None of the tools require more than 7.

READWRITE access may cause problems. Both ‘ed’ and ‘msg’ use this access on their scratch buffer files. If necessary, you may have to implement these functions by opening the file twice, one at READ and once at WRITE access.

SEE ALSO

create(2), close(2)

DIAGNOSTICS

‘open’ returns ERR if the file does not exist, if the file is not readable/writable or if too many files are open.

NAME

Opendr - open directory for reading filenames

SYNOPSIS

```
filedes function opendr( name, fd))

character name(FILENAMESIZE)
filedes fd
```

DESCRIPTION

'opendr' opens the specified directory for READ access via 'gdrprm'. All write access to directories is implicitly done with the 'amove', 'create' and 'remove' primitives. 'name' is a character string representing the directory as a pathname or whatever format is expected by the local operating system. The name is passed as a character array terminated by an EOS character.

'fd' is a "filedes" descriptor for use in other directory manipulation primitives. The value returned by the function is the value of 'fd' or ERR.

IMPLEMENTATION

'opendr' is the directory equivalent to 'open' at READ access. It prepares the directory for sequential access to the filenames stored there.

SEE ALSO

closdr(2), gdrprm(2), gdraux(2), amove(2), create(2), remove(2)

DIAGNOSTICS

A value of ERR is returned if the directory could not be opened, or too many directories have already been opened.

NAME
    Prompt - get next line from file, prompting if a terminal

SYNOPSIS
```
    integer function prompt( pstr, line, fd)

    character pstr(ARB), line(MAXLINE)
    filedes fd
```

DESCRIPTION
    'prompt' is identical to 'getlin', with the exception that if 'fd' corresponds to a terminal unit, the prompt string 'pstr' will be displayed before the read is performed. The line read will be placed in 'line' with the possible return values being identical with those of 'getlin'. If 'fd' does not correspond to a terminal unit, 'prompt' simply performs a 'getlin'.

    There is no implicit <CARRIAGE-RETURN, LINEFEED> pair at the end of the prompt string (otherwise, there would be no need for this primitive). If embedded NEWLINES are found in the prompt string, they should result in <CARRIAGE-RETURN, LINEFEED> pairs being output on the terminal at the appropriate locations.

    If after outputting the prompt string and reading the line 'prompt' sees the NEWLINE character preceeded by an '@', the '@' should be replaced by a BLANK, a secondary prompt string consisting of pstr(1) followed by an '_' (underscore) should be displayed, and another line fetched into the buffer after the inserted BLANK. This process should be repeated until the NEWLINE is not escaped, or MAXLINE characters have been fetched.

IMPLEMENTATION
    The output of the prompt string is conditionalized upon
```
                            isatty(fd) == YES
```
    Since 'fd' generally is associated with an 'open' at READ access, 'prompt' may have to temporarily open a unit to the terminal at WRITE access in order to display the prompt string.

SEE ALSO
    putlin(2)

DIAGNOSTICS
    none

NAME
    Ptrcpy - copy linepointers

SYNOPSIS
```
    subroutine ptrcpy( in, out)

    linepointer in, out
```

DESCRIPTION
    ‘ptrcpy’ copies the linepointer variable ‘in’ to the linepointer variable ‘out’.

IMPLEMENTATION

SEE ALSO
    ptreq(2), note(2), seek(2)

DIAGNOSTICS
    none

NAME

Ptreq - determine if two linepointers are equal

SYNOPSIS

```
integer function ptreq( ptr1, ptr2)

linepointer ptr1, ptr2
```

DESCRIPTION

'ptreq' checks for the equality of the two linepointers passed as 'ptr1' and 'ptr2'. If they are equal, the value YES is returned, otherwise NO is returned.

IMPLEMENTATION

SEE ALSO

ptrcpy(2), note(2), seek(2)

DIAGNOSTICS

none

NAME

Ptrtoc - format linepointer into characters

SYNOPSIS

```
integer function ptrtoc( ptr, buf, size)

linepointer ptr
character buf(size)
integer size
```

DESCRIPTION

'ptrtoc' generates a printable character string which represents the value of the linepointer variable 'ptr'. The characters are placed in the buffer 'buf'. If the formatted buffer would exceed 'size' characters (including the EOS), only 'size' characters are placed in 'buf'. The length of the formatted string is returned as the value of the function.

IMPLEMENTATION

SEE ALSO

ptreq(2), ptrcpy(2), note(2), seek(2), ctoptr(2)

DIAGNOSTICS

none

NAME
    Putch - write character to file

SYNOPSIS
```
    subroutine putch( c, fd)

    character c
    filedes fd
```

DESCRIPTION
    'putch' writes the character 'c' onto the file specified by 'fd'. If 'c' is the NEWLINE character, the appropriate action is taken to indicate the end of the record on the file. The character is assumed to be in ASCII format; if the external representation is not ASCII, the necessary conversion is done.

    If fd' corresponds to a RAW or RARE terminal unit, the character 'c' is immediately written to the terminal with no interpretation by the native operating system's terminal handler.

IMPLEMENTATION
    Interspersed calls to 'putch' and 'putlin' should work properly.

SEE ALSO
    putlin(2), getch(2), getlin(2), stmode(2)

DIAGNOSTICS
    If an error occurs when a record is flushed, an ugly error message will appear on your terminal.

NAME
Putlin - output a line onto a given file

SYNOPSIS
```
subroutine putlin( line, fd)

character line(ARB)
filedes fd
```

DESCRIPTION
'putlin' outputs the character array 'line' onto the file specified by 'fd'. 'line' is an ASCII character array terminated with an EOS character. NEWLINE characters are permitted in the array, with the effect of flushing the record since the last NEWLINE character. If none is specified, no carriage-return (or end-of-record) is assumed. If the external representation is not ASCII, translation occurs before writing the record.

If 'fd' is a RAW or RARE mode terminal unit, the 'line' buffer is written immediately to the terminal, with no interpretation by the terminal driver.

IMPLEMENTATION
Interspersed calls to 'putch' and 'putlin' are permitted. A common implementation for COOKED mode units is to have 'putlin' call 'putch' until an EOS character is found.

SEE ALSO
putch(2), getch(2), getlin(2), stmode(2)

DIAGNOSTICS
none

NAME

Readf - read from an opened file

SYNOPSIS

```
count = readf( buf, n, fd)

character buf(ARB)
integer n
filedes fd
integer count returned as count/EOF
```

DESCRIPTION

Readf reads 'n' bytes from the file opened on file descriptor 'fd' into the array 'buf'. The bytes are placed in 'buf' one per array element. Readf is the typical way of doing binary reads on files. Readf returns the number of bytes actually read. In most cases, this is equal to 'n'. However, it may be less if an EOF has been encountered or if 'fd' specified a device such as a terminal where less than 'n' bytes were input.

IMPLEMENTATION

Readf is the typical way of implementing binary I/O. Do whatever is necessary on your system to allow users to get at the file directly.

If reasonable, design readf to work properly in conjunction with getch and getlin.

SEE ALSO

writef(2), getch(2), putch(2)

DIAGNOSTICS

none

NAME

Remark - print single-line message

SYNOPSIS

```
subroutine remark(messag)

character messag(ARB)
```

DESCRIPTION

'remark' writes the message onto the standard error file ERROUT. A NEWLINE is always generated, even though one may not appear in the message.

The 'messag' array is generally a Fortran hollerith string in the format generated by the Ratfor quoted string capability. It may also be an character array terminated with an EOS character.

IMPLEMENTATION

If a quoted string is used as the argument to remark, it should, by convention, be terminated by a PERIOD ('.'). This permits all implementations to locate the end of the string to print. If a NEWLINE character is not found at the end of the string, one must be 'putch'ed to ERROUT.

SEE ALSO

putlin(2)

DIAGNOSTICS

none

NAME

Remove - delete a file from the file system

SYNOPSIS

```
integer function remove(name)

character name(FILENAMESIZE)

return(OK/ERR)
```

DESCRIPTION

'remove' deletes a file from the file system when invoked from a running program. 'name' is a character string representing a pathname or filename in whatever format is used by the local operating system. It is passed as a character array terminated by an EOS character.

The function value returned should be OK/ERR depending upon the success of the delete operation. Deletion of a non-existent file should result in a return of OK.

IMPLEMENTATION

The file to be removed need not be opened before remove is called. If the file is currently open on other units, remove should display an error message.

SEE ALSO

open(2), close(2), create(2)

DIAGNOSTICS

If an error occurs, a value of ERR is returned.

NAME
    Scratf - generate unique scratch file name

SYNOPSIS
```
    subroutine scratf( seed, name)

    character seed(ARB), name(FILENAMESIZE)
```

DESCRIPTION
    'scratf' is used to generate unique scratch file names. 'seed' is passed as a character array terminated by an EOS character, and will be used to make this scratch file name unique with respect to other scratch files generated by this process. The scratch file name generated is returned in 'name' as an EOS-terminated character array. Only the first three (3) characters of 'seed' are guaranteed to be used, so the user should be sure that all 'seed's used in the program are unique in the first three characters.

IMPLEMENTATION
    'scratf' is used to avoid conflicts which occur when more than one user is logged in under a single user or directory name. The optimal implementation would be to return an absolutely unique file name based upon 'seed', which can often be achieved via some manipulation of the process name or id. It is common practice to have all scratch files generated by the tools reside in a common scratch file directory.

    On single-user systems or systems which support the notion of "local files", 'scratf' can simply return 'seed' as 'name'.

SEE ALSO
    getdir(2)

DIAGNOSTICS
    If the file name could not be generated, a message should be printed.

NAME

Seek - move read/write pointer

SYNOPSIS

```
subroutine seek( addres, fd)

linepointer addres
filedes fd
```

DESCRIPTION

'seek' positions the file specified by 'fd' for a subsequent read or write beginning at 'addres'. 'addres' is a variable of type linepointer containing the system-dependent address of the record, which was originally obtained by a call to 'note'.

If a write is performed after a 'seek', the file is truncated after that line, due to the sequential nature of the Tool's I/O.

IMPLEMENTATION

'seek' is generally used on files opened at READWRITE access. The units of 'addres' are chosen to be whatever is most appropriate for the system involved.

SEE ALSO

note(2), ptrcpy(2), ptreq(2)

DIAGNOSTICS

none

NAME
    Sleep - stop process for period of time

SYNOPSIS
```
    subroutine sleep(secnds)

    integer secnds
```

DESCRIPTION
    'sleep' causes the current process to suspend itself for the period of time specified in the parameter 'secnds'. Control resumes at the next instruction after the call sleep statement when the time period has elapsed.

IMPLEMENTATION
    The only utility which uses 'sleep' is 'sched'. Therefore, it is not necessary, although such a facility will make many real-time tasks easier to solve

SEE ALSO

DIAGNOSTICS
    none

NAME

Spawn - initiate sub-process

SYNOPSIS

```
integer function spawn( image, args, pid, waitfl)

character image(FILENAMESIZE), args(ARGBUFSIZE), pid(PIDSIZE), waitfl
```

DESCRIPTION

'spawn' causes the initiation of a sub-process. 'image' is an EOS-terminated character array specifying the filename of the image to be initiated, in either pathname or local file format.

'args' is a character array specifying the command line to be passed to the sub-process. The name by which the image was invoked should be the first word in the argument buffer. If the string passed in 'image' is equal to the string "local", then 'args' should contain the native command line to be passed to the local command language interpreter.

If 'waitfl' == WAIT & equal(image, "local") == NO, 'spawn' should scan 'args' for redirection of STDOUT and ERROUT. If either of these units are not redirected, the corresponding unit should be closed, and an APPEND redirection to that file should be formatted into 'args' for the child. When the child process completes, the unit should be re-opened at APPEND access, thus permitting the correct interleaving of output on these units between parent and child processes.

'pid' is an array to receive the process id of the spawned sub-process. This id may then be used in other process control primitives.

'waitfl' is a flag indicating whether the parent process wishes to synchronize its execution with the termination of the sub-process. If the value of WAIT is specified, 'spawn' will not return control until the sub-process has completed. If NOWAIT is specified, 'spawn' immediately returns to the caller (for use with real pipes). Processes spawned with this flag are required to exit when the parent process exits. If BACKGR is specified, the sub-process is spawned in the background and control is immediately returned to the caller. Background process come to life with the standard I/O units directed to the null device, and have an existence totally independent of that of the parent. It is common to have the background processes run at a lower priority than foreground processes.

If an error occurs during the initiation of the sub-process, ERR is returned to the user. If the sub-process abnormally exits when WAIT has been specified, a value of CHILD_ABORTED is returned. Otherwise, OK is returned.

IMPLEMENTATION

'spawn' is normally the most difficult primitive to implement. A few of the major obstacles which must be overcome are:

1. Does the operating system permit a running process to spawn a sub-process? If it provides a multi-user, interactive environment, it most certainly could, but it may not be common knowledge as to how to do it.

2. Once one has determined how to spawn the process, it is necessary to determine how to control it. If the operating system does not provide any synchronization methods, they must be implemented.

3. Finally, one must determine how to communicate the arguments and environment information to the sub-process. This generally entails an exploration of the system-provided interprocess-communication mechanisms, and often requires the invention of better ones.

SEE ALSO

DIAGNOSTICS

A value of ERR is returned if an error occurs during sub-process initiation. If the sub-process exits abnormally when 'waitfl' had a value of WAIT, CHILD_ABORTED is returned.

NAME

Stmode - change mode on terminal unit to RAW/RARE/COOKED

SYNOPSIS

```
integer function stmode( fd, mode)

filedes fd
integer mode
```

DESCRIPTION

'stmode' is used to change an open ratfor unit, 'fd', to a different mode of operation, as specified by 'mode'. The default mode when a unit is opened or created is COOKED. If the unit corresponds to an interactive device, it is permitted to change the mode to RARE or RAW. The value to which the mode is set is returned as the value of the function.

IMPLEMENTATION

'stmode' usually sets a flag for the particular unit, such that future 'getch' and 'putch' calls on the unit will be performed correctly for the given mode of operation. If the unit specified is not currently associated with a particular file, the value of ERR will be returned.

SEE ALSO

getch(2), putch(2)

DIAGNOSTICS

If the unit is not currently associated with a file, the value of ERR is returned.

NAME
    Symbols - standard symbol definitions

```
#================= GENERAL SYMBOL DEFINITIONS =================

# General definitions for software tools
# Should be put on a file named 'symbols'
# Used by all the tools; read automatically by preprocessor


#   Many of these symbols may change for your particular machine.
#   The values provided are intended as guidelines, and may
#   well serve you adequately, but don't hesitate to change them if
#   necessary.

# In particular, the following might have to change for your system:
#         TERMINAL_IN
#         TERMINAL_OUT
#         MAXLINE
#         FILENAMESIZE
#         DRIVER    and    DRETURN
#         MAXOFILES
#         character

#   Also, watch out for the following definitions, which
#   may conflict with the Fortran operators on your system:
#       AND         OR        NOT


#  Many of the definitions will be used in character variables.
#  They must be defined to be something other than a valid ascii
#  character--such as a number > 255 or a negative number.
#  If you have defined "character" to be "integer", then you may
#  use either a very large number or a small negative number.
#  If you have defined "character" to be something like an 8-bit
#  signed field, you'll need to use negative numbers.
#  Use of a standard integer (whatever is the default size on your
#  machine) is STRONGLY recommended, despite the apparent waste of
#  storage.


# The following constants affect conditional pre-processing

define(VAX_VMS,)                # Define CPU and Operating system.
define(LARGE_ADDRESS_SPACE,)    # this is defined if the user has at least
                                # 18 address bits for use
define(TREE_STRUCT_FILE_SYS,)   # this is defined is the file system is
                                # tree structured
define(SORTED_DIRECTORIES,)     # defined if the directories are inherently
                                # sorted
# ASCII control character definitions:

define(NUL,0)
define(SOH,1)
define(STX,2)
define(ETX,3)
define(EOT,4)
define(ENQ,5)
define(ACK,6)
define(BEL,7)
define(BS,8)
define(HT,9)
define(LF,10)
define(VT,11)
define(FF,12)
define(CR,13)
define(SO,14)
define(SI,15)
define(DLE,16)
define(DC1,17)
define(DC2,18)
define(DC3,19)
define(DC4,20)
define(NAK,21)
define(SYN,22)
define(ETB,23)
define(CAN,24)
define(EM,25)
define(SUB,26)
define(ESC,27)
define(FS,28)
define(GS,29)
define(RS,30)
define(US,31)
define(SP,32)
define(DEL,127)

# Synonyms for ASCII control characters

define(BACKSPACE,8)
define(BELL,7)
define(BLANK,32)
define(CARRIAGE_RETURN,13)
define(NEWLINE,10)
define(RUBOUT,127)
define(TAB,9)

# Printable ASCII characters:
```

```
define(ACCENT,96)
define(AMPER,38)         # ampersand
define(AMPERSAND,AMPER)
define(AND,AMPER)
define(ATSIGN,64)
define(BACKSLASH,92)
define(BANG,33)          # exclamation mark
define(BAR,124)
define(BIGA,65)
define(BIGB,66)
define(BIGC,67)
define(BIGD,68)
define(BIGE,69)
define(BIGF,70)
define(BIGG,71)
define(BIGH,72)
define(BIGI,73)
define(BIGJ,74)
define(BIGK,75)
define(BIGL,76)
define(BIGM,77)
define(BIGN,78)
define(BIGO,79)
define(BIGP,80)
define(BIGQ,81)
define(BIGR,82)
define(BIGS,83)
define(BIGT,84)
define(BIGU,85)
define(BIGV,86)
define(BIGW,87)
define(BIGX,88)
define(BIGY,89)
define(BIGZ,90)
define(CARET,94)
define(COLON,58)
define(COMMA,44)
define(DASH,45)          #same as MINUS
define(DIG0,48)
define(DIG1,49)
define(DIG2,50)
define(DIG3,51)
define(DIG4,52)
define(DIG5,53)
define(DIG6,54)
define(DIG7,55)
define(DIG8,56)
define(DIG9,57)
define(DOLLAR,36)
define(DQUOTE,34)
define(EQUALS,61)
define(ESCAPE,ATSIGN)           # escape char for ch, find, tr, ed, and sh.
define(GREATER,62)
define(LBRACE,123)
define(LBRACK,91)
define(LESS,60)
define(LETA,97)
define(LETB,98)
define(LETC,99)
define(LETD,100)
define(LETE,101)
define(LETF,102)
define(LETG,103)
define(LETH,104)
define(LETI,105)
define(LETJ,106)
define(LETK,107)
define(LETL,108)
define(LETM,109)
define(LETN,110)
define(LETO,111)
define(LETP,112)
define(LETQ,113)
define(LETR,114)
define(LETS,115)
define(LETT,116)
define(LETU,117)
define(LETV,118)
define(LETW,119)
define(LETX,120)
define(LETY,121)
define(LETZ,122)
define(LPAREN,40)
define(MINUS,45)
define(NOT,BANG)        # used in pattern matching; choose ~, ^, or !
define(OR,BAR)
define(PERCENT,37)
define(PERIOD,46)
define(PLUS,43)
define(QMARK,63)
define(RBRACE,125)
define(RBRACK,93)
define(RPAREN,41)
define(SEMICOL,59)
define(SHARP,35)
define(SLASH,47)
define(SQUOTE,39)
define(STAR,42)
define(TAB,9)
define(TILDE,126)
define(UNDERLINE,95)


# Ratfor language extensions:

define(andif,if)
define(ARB,1000)
define(character,logical*1)     # define character data type
define(CHARACTER,character)
define(DS_DECL,integer $1($2);character c$1(arith($2,*,CHAR_PER_INT));
equivalence (c$1(1),$1(1));common/cdsmem/$1)
define(PB_DECL,integer pbp, pbsize; character pbbuf($1);
common/cpback/pbp, pbsize, pbbuf)
define(cvt_to_cptr,(CHAR_PER_INT*($1-1)+1))     # convert pointer to char ptr
define(elif,else if)
define(filedes,integer)         # file descriptor/designator data type
define(FILEDES,filedes)
define(IS_DIGIT,(DIG0<=$1&$1<=DIG9))    # valid only for ASCII!
define(IS_LETTER,(IS_UPPER($1)|IS_LOWER($1)))
define(IS_LOWER,(LETA<=$1&$1<=LETZ))
define(IS_UPPER,(BIGA<=$1&$1<=BIGZ))
define(long_real,double precision)
define(linepointer,real*8)
define(NULLPOINTER,-1)
define(LINEPTRSIZE,MAXCHARS)
define(pointer,integer)
define(POINTER,integer)



# Input/output modes:

define(APPEND,4)
define(PRINT,5)         # print file access
define(READ,1)
define(READWRITE,3)
define(WRITE,2)


# Standard input/output ports:

define(ERROUT,3)        # standard error file
define(STDERR,ERROUT)
define(STDIN,1)         # standard input file
define(STDOUT,2)        # standard output file


# TERMINAL_IN and TERMINAL_OUT are the names of the I/O channels
# from and to the user's terminal, respectively.  It's highly likely
# there is no such thing on your system; in this case, simply invent
# a name that is not likely to conflict with any file name.
# For example, the VAX/VMS version of the tools uses "TT", the RSX/11M
# version uses "TI:", the DEC 10 version uses "tty:", and the Prime
# version uses "/dev/tty".
# Note that you must make the 'open' primitive recognize this name
# and provide access to the terminal accordingly.

define(TTY_NAME,"TT")
define(TERMINAL_IN,TTY_NAME)
define(TERMINAL_OUT,TTY_NAME)


# Manifest constants included for readability and modifiability:

define(ALPHA,-9)
define(ASCII,12)                # flag for ascii character file
define(BEGINNING_OF_FILE,-2)    # flag to seek for positioning at
                                # the beginning of a file
define(BINARY,60)               # flag for indicating binary file
define(CHILD_ABORTED,101)       # possible status return from spawn
define(DIGIT,2)
define(END_OF_FILE,-1)          # flag to seek for positioning at
                                # end of file
define(EOF,-1)
define(EOS,0)
define(ERR,-3)
define(HUGE,30000)              # some arbitrarily large number
define(LAMBDA,0)                # end of list marker
define(LETTER,1)
define(LOCAL,6)                 # flag for local-type character file
define(NO,0)
define(NOERR,0)                 # flag for successful completion
define(OK,0)                    # success flag
define(PATH,5)                  # type == PATH
define(TMO,-4)                  # error return for timeout (dpm 8-Jun-81)
define(USERSIZE,20)             # size of username returned by userid
define(YES,1)


# Size limiting definitions for important objects:

define(FILENAMESIZE,100)        # max characters in file name
                                # (including EOS)
define(MAXARG,MAXLINE)          # max size of command line argument
define(MAXARGS,25)              # some tools require this for max no of args
define(MAXCHARS,20)             # max nbr of chars when converting
                                # from integers to characters
                                # (used by putint, outnum, etc.)
define(MAXLINE,512)             # normal size of line buffers;
                                # must be at least 1 more than MAXCARD
```

```
define(MAXCARD,arith(MAXLINE,-,1))
define(MAXNAME,FILENAMESIZE)    # max size of file name
define(MAXOFILES,15)            # max nbr opened files allowed at a time
define(MAXPAT,128)              # max size of encoded patterns
                                # (used in string matching)
define(NCHARS,33)               # number of special characters


# Machine-dependent parameters: (VAX)

define(BITS_PER_CHAR,8)
define(BITS_PER_WORD,32)        # (dpm 8-Jun-81)
define(CHARS_PER_WORD,4)        # (dpm 8-Jun-81)
define(CHAR_PER_INT,4)
define(MAX_INTEGER,1073241823)  # 2**30 - 1 (dpm 8-Jun-81)
define(MIN_INTEGER,-1073241824) # -2**30 - 1 (dpm 8-Jun-81)
define(MAX_REAL_EXP,38)
define(MIN_REAL_EXP,-37)        # (dpm 8-Jun-81)
define(REAL_PRECISION,7)        # (dpm 8-Jun-81)


# DRIVER is defined as those things you need to do to start a Software
# Tools program running.  The following is a common approach, but you
# may have to change it (for example, by adding a "program" card).
# Many machines will require no special driver procedure other than
# the call to 'initst'.

define(DRIVER,subroutine main # $1)

# DRETURN is used to finish up a Software Tools program:

define(DRETURN,return)          # (returning from subroutine defined in DRIVER)


# Definitions for 'spawn' primitive (if implemented):

define(WAIT,LETW)               # wait for subprocess to complete
define(NOWAIT,LETN)             # control returns as soon as
                                # subprocess starts
define(BACKGR,LETB)             # spawning a background process
define(PIDSIZE,9)
define(ARGBUFSIZE,512)


# rawmode io definitions

define(COOKED,0)                # line-at-a-time (record) io
define(RAW,1)                   # char-at-a-time (unfiltered) io
define(RARE,2)                  # char-at-a-time (with interrupts) io
```

```
# definitions for obtaining directory strings

define(BINDIRECTORY,1)
define(USRDIRECTORY,2)
define(TMPDIRECTORY,3)
define(LPRDIRECTORY,4)
define(MSGDIRECTORY,5)
define(MAILDIRECTORY,5)
define(MANDIRECTORY,6)
define(SRCDIRECTORY,7)
define(INCDIRECTORY,8)
define(LIBDIRECTORY,9)

# definitions needed for directory operations

define(TCOLWIDTH,24)            # width of date string returned by gdraux
define(MAXDIRECTS,10)           # max number of path fields in file spec

# definitions needed for double integer manipulations

define(initdi,{$1(1) = 0; $1(2) = 0})
define(incrdi,{$1(2) = $1(2) + 1; if($1(2) >= 10000)
{$1(1) = $1(1) + 1; $1(2) = 0}})
define(decrdi,{$1(2) = $1(2) - 1; if($1(2) < 0)
{$1(1) = $1(1) - 1; $1(2) = 9999}})
define(adddi,{$2(1) = $2(1) + $1(1); $2(2) = $2(2) + $1(2);
if ($2(2) >= 10000){$2(1) = $2(1) + 1; $2(2) = $2(2) - 10000}})
define(subdi,{$2(1) = $2(1) - $1(1); $2(2) = $2(2) - $1(2);
if ($2(2) < 0){$2(1) = $2(1) - 1; $2(2) = $2(2) + 10000}})

# It may be necessary to add special definitions; for example
# names of important directories, substitute routine names for
# Software Tools primitives that conflict with local subprograms,
# etc.

define(putc,putch($1,STDOUT))
define(getc,ifelse($1,,getch,getch($1,STDIN)))
define(putdec,putint($1,$2,STDOUT))
define(index,indexx)
define(INDEX,index)
define(SS_NORMAL,1)
define(BOTH_SUFFIX,".sh@e.exe@e@n")
define(IMAGE_SUFFIX,".exe@e@n")
define(NO_SUFFIX,"@e@n")
define(mkuniq,scratf)
# special definitions for pwait
define(ANDWAIT,51)
define(ORWAIT,50)
define(TERMSGSIZE,21)
```

NAME

Trmlst - list terminal a user is logged onto

SYNOPSIS

```
integer function trmlst(user, tlist)

character user(ARB), tlist(ARB)

return(number of terminals found)
```

DESCRIPTION

'trmlst' scans the system for the names of all terminals upon which 'user' is logged onto, and returns the names as blank-separated tokens in the array 'tlist'. The number of terminals found is returned as the value of the function.

IMPLEMENTATION

As for 'brdcst', this may be a difficult function to provide. It may be safely implemented as a stub returning 0. It is only used by 'sndmsg' and 'mail' to notify users of mail.

SEE ALSO

brdcst(2)

DIAGNOSTICS

Returns 0 if the user is not currently logged in.

NAME

Writef - write to an opened file

SYNOPSIS

```
count = writef( buf, n, fd)

character buf(ARB)
integer n
filedes fd
integer count returned as count/ERR
```

DESCRIPTION

Writef writes 'n' bytes from the array 'buf' to the file opened on file descriptor 'fd'. Writef is the typical way of doing binary writes to files.

Writef returns the number of bytes actually written. In most cases, this is equal to 'n'. If, however, a write error occurs, writef returns ERR.

IMPLEMENTATION

Writef is the typical way of implementing binary I/O. Do whatever is necessary on your system to allow users to get at the file directly.

If reasonable, design writef to work properly in conjunction with putch and putlin.

SEE ALSO

readf(2), putch(2), putlin(2)

DIAGNOSTICS

none

```
 ____            _   _               _____
/ ___|  ___  ___| |_(_) ___  _ __   |___ /
\___ \ / _ \/ __| __| |/ _ \| '_ \    |_ \   _____
 ___) |  __/ (__| |_| | (_) | | | |  ___) | |_____|
|____/ \___|\___|\__|_|\___/|_| |_| |____/

 _     _ _                             ____             _   _
| |   (_) |__  _ __ __ _ _ __ _   _   |  _ \ ___  _   _| |_(_)_ __   ___  ___
| |   | | '_ \| '__/ _` | '__| | | |  | |_) / _ \| | | | __| | '_ \ / _ \/ __|
| |___| | |_) | | | (_| | |  | |_| |  |  _ < (_) | |_| | |_| | | | |  __/\__ \
|_____|_|_.__/|_|  \__,_|_|   \__, |  |_| \_\___/ \__,_|\__|_|_| |_|\___||___/
                              |___/
```

NAME

Acopy - copy n characters from one file to another

SYNOPSIS

```
    subroutine acopy(ifd, ofd, n)

    integer n
    filedes ifd, ofd
```

DESCRIPTION

'acopy' copies 'n' characters from 'ifd' to 'ofd', both of which are assumed open. If an EOF is encountered on 'ifd' before 'n' characters have been copied, the routine simply returns.

SEE ALSO

DIAGNOSTICS

none

NAME
    Adddi - add double integers together

SYNOPSIS
```
    adddi(dbl1,dbl2)

    integer dbl1(2), dbl2(2)

    expands into:

          {
          dbl2(1) = dbl2(1) + dbl1(1)
          dbl2(2) = dbl2(2) + dbl1(2)
          if (dbl2(2) >= 10000)
            {
            dbl2(1) = dbl2(1) + 1
            dbl2(2) = dbl2(2) - 10000
            }
          }
```

DESCRIPTION
    Invocation of this macro causes the first double integer argument to be added to the second. If a carry is necessary, it is performed. See the entry for ‘initdi’ for more information on double integers.

SEE ALSO
    initdi(3), incrdi(3), decrdi(3), subdi(3)

DIAGNOSTICS

NAME

Addset - put c in array(j) if it fits, increment j

SYNOPSIS

```
stat = addset(c, array, j, maxsize)

character c, array(ARB)
integer j                  # j is incremented
integer maxsize
integer stat returned as YES/NO
```

DESCRIPTION

Adds a character at a time to a specified position of an array and increments the index. It also checks that there's enough room to do so.

The array is an ascii character array stored one character per word. 'c' is a single ascii character.

YES is returned if the routine succeeded, otherwise NO.

SEE ALSO

scopy(3), stcopy(3), concat(3)

DIAGNOSTICS

None

NAME

Addstr - add string s to str(j) if it fits, increment j

SYNOPSIS

```
    stat = addstr(s, str, j, maxsize)

    character s(ARB), str(ARB)
    integer j                 # j is incremented
    integer maxsize
    integer stat returned as YES/NO
```

DESCRIPTION

Copies the string 's' to array 'str', starting in location 'j'. `j' is incremented to point to the next free position in 'str'.

If the addition of `s' to `str' will exceed its maximum length (maxsize), no copying is done and the status NO is returned.

Both `s' and `str' are ascii character arrays stored one character per array element.

YES is returned if the routine succeeded, otherwise NO.

SEE ALSO

scopy(3), stcopy(3), addset(3), concat(3)

DIAGNOSTICS

None

NAME

    Adrfil - get name of user-information database file

SYNOPSIS

```
    subroutine adrfil(file)

    character file(FILENAMESIZE)
```

DESCRIPTION

    'adrfil' returns the local file specification for the user-information database file, known as "~msg/address".

SEE ALSO

    mailid(2), homdir(2)

DIAGNOSTICS

    none

NAME

Agetch - get next character from an archive module

SYNOPSIS

```
character function agetch(c, fd, size)

character c
filedes fd
integer size(2)
```

DESCRIPTION

'agetch' fetches the next character from the archive module opened on 'fd' and returns it in the variable 'c' and as the value of the function. The 'size' argument is that returned by an 'aopen' or 'agethd' call, and is decremented by 'agetch' to reflect the decrease in size of the remainder of the module. If the end of the module is detected, or a true end of file is detected on 'fd', the value EOF is returned.

SEE ALSO

agethd(3), agtlin(3), aopen(3), askip(3)

DIAGNOSTICS

Returns EOF if end of module is detected.

NAME

Agethd - get next archive header from file

SYNOPSIS

```
integer function agethd(fd, buf, size, fsize)

filedes fd
character buf(MAXLINE)
integer size(2), fsize(2)
```

DESCRIPTION

'agethd' reads the next line from the archive module represented by the file descriptor 'fd' and the size 'fsize'. If the line is of the form of an archive header, the name of the module is placed in 'buf', and the size of the module is placed in 'size', with 'fsize' decremented to represent the decrease in size of the containing module. The value OK is returned if successful. If an end of module is detected, the value EOF is returned, and if the line read is not of the proper format, a value of ERR is returned.

SEE ALSO

agetch(3), agtlin(3), aopen(3), askip(3)

DIAGNOSTICS

Returns EOF on end of module and ERR if improper archive format.

NAME
    Agtlin - get next line from an archive module

SYNOPSIS
```
    integer function agtlin(buf, fd, size)

    character buf(MAXLINE)
    filedes fd
    integer size(2)
```

DESCRIPTION
    'agtlin' fetches the next line of input from the archive module represented by the arguments 'fd' and 'size'. If another line is found, it is placed in 'buf', 'size' is decremented by the number of characters in the line, and this number is returned as the value of the function. If an end of module is detected, a value of EOF is returned.

SEE ALSO
    agetch(3), agethd(3), aopen(3), askip(3)

DIAGNOSTICS
    Returns EOF if end of module is detected.

NAME

Alldig - determine if string is all digits

SYNOPSIS

```
    integer function alldig(str)

    character str(ARB)
```

DESCRIPTION

'alldig' determines if the given string is all digits. If this is true, the value YES is returned, otherwise, NO.

SEE ALSO

type(3)

DIAGNOSTICS

A value of NO is returned if not all digits.

NAME

Amatch - look for pattern matching regular expression

SYNOPSIS

```
    integer function amatch(line, from, pat, tagbeg, tagend)

    character line(ARB)
    integer   from, pat(MAXPAT)
    integer   tagbeg(10), tagend(10)
              (element "i+1" returns start or end, respectively,
              of "i"th tagged sub-pattern)
```

DESCRIPTION

Amatch scans 'line' starting at location 'from', looking for a pattern which matches the regular expression coded in 'pat'. If the pattern is found, the next available location in `line' is returned. If the pattern is not found, amatch returns 0.

The regular expression in 'pat' must have been previously encoded by 'getpat' or 'makpat'. (For a complete description of regular expressions, see the writeup on the editor.)

Amatch is a special-purpose version of match, which should be used in most cases.

SEE ALSO

match(3), getpat(3), makpat(3), ed(1)

DIAGNOSTICS

A value of 0 is returned if the pattern does not match.

NAME

Aopen - open archive module for reading

SYNOPSIS

```
filedes function aopen(name, fd, size)

character name(FILENAMESIZE)
filedes fd
integer size(2)
```

DESCRIPTION

'aopen' opens the archive module specified in 'name' for reading with subsequent calls to 'agetch' and 'agtlin'. If the open is successful, the resulting file descriptor is placed in 'fd', as well as returned as the function value; the size of the module is placed in the variable 'size'. Failure is signalled by returning a value of ERR.

The format of the name specification is quite straight-forward; the syntax is:

```
                      filename['module]...
```

If no module names are specified, 'aopen' is equivalent to an 'open' call at READ access, and an "infinite" module size is placed in 'size'.

EXAMPLES

```
character c
character agetch
integer size(2)
filedes fd
filedes aopen

string name "rlib.w'lib.r'arsubs.r'aopen"

if (aopen(name, fd, size) == ERR)
  call cant(name)
while (agetch(c, fd, size) != EOF)
  call putch(c, STDOUT)
call close(fd)
```

SEE ALSO

agetch(3), agethd(3), agtlin(3), askip(3)

DIAGNOSTICS

Returns ERR if the specified archive module cannot be opened.

NAME
    Argtab - fetch tab information from command line

SYNOPSIS
```
    subroutine argtab(buf)

    character buf(MAXLINE)
```

DESCRIPTION
    'argtab' reads the command line arguments, using 'getarg', and copies those arguments which 'detab' and 'entab' understand into 'buf', separated by blank characters. This is in preparation for calling 'settab' to set the TAB stops.

SEE ALSO
    settab(3), detab(1), entab(1), getarg(2)

DIAGNOSTICS
    none

NAME

Askip - skip rest of archive module contents

SYNOPSIS

```
subroutine askip(fd, size, fsize)

filedes fd
integer size(2), fsize(2)
```

DESCRIPTION

'askip' skips the number of characters indicated by 'size' in the archive module specified by 'fd' and 'fsize'. 'fsize' is decreased by the number of characters skipped. This routine is handy when using 'aopen' to open a nested archive, and then scanning the archive modules at that level for the ones of interest.

EXAMPLES

```
character buf(MAXLINE)
integer size(2), fsize(2)
integer agethd
filedes fd
filedes aopen

string name "rlib.w`lib.r"
string line "The modules contained in rlib.w`lib.r are:@n"

if (aopen(name, fd, fsize) == ERR)
  call cant(name)
call putlin(line, STDOUT)
while (agethd(fd, buf, size, fsize) == OK)
  {
  call putlin(buf, STDOUT)
  call putch('@n', STDOUT)
  call askip(fd, size, fsize)
  }
call close(fd)
```

SEE ALSO

agetch(3), agethd(3), agtlin(3), aopen(3)

DIAGNOSTICS

NAME
    Badarg - output "invalid argument" message

SYNOPSIS
    subroutine badarg(arg)

    character arg(ARB)

DESCRIPTION
    'badarg' displays the following message on Error Output:

```
              ? Ignoring invalid argument '<arg>'
```

    where <arg> is replaced by the contents of 'arg'.

SEE ALSO
    getarg(2)

DIAGNOSTICS
    none

NAME
    Bubble - bubble sort integers

SYNOPSIS

```
    subroutine bubble(v, n)

    integer n, v(n)
```

DESCRIPTION
    'bubble' performs a bubble sort on the integers v(1) ... v(n). As is well known, the bubble sort algorithm should only be used for very small arrays. If larger arrays need to be sorted, see the entry on 'shell'.

SEE ALSO
    shell(3)

DIAGNOSTICS
    none

NAME
    Cant - print "Can’t open" message and terminate execution

SYNOPSIS
    call cant(name)

    character name(ARB)

DESCRIPTION
    Prints an error message (on ERROUT) indicating file "name" could not be opened. All open files are closed and execution is terminated. Name is an ascii character array terminated with an EOS marker.

SEE ALSO
    error(3), remark(2)

DIAGNOSTICS
    None

NAME
    Catsub - add replacement text to new buffer

SYNOPSIS
```
    subroutine catsub(lin, from, to, sub, new, k, maxnew)

    integer from, to, k, maxnew
    character lin(MAXLINE), new(maxnew), sub(ARB)
```

DESCRIPTION
    The string represented by lin(from) ... lin(to-1) is replaced according to the instructions in 'sub'(which has been generated via a call to 'getsub' or 'maksub'); the replacement text is appended to 'new' starting at position 'k'. 'k' is incremented as the substitutions are added, and points to the EOS location 'new' upon return. 'maxnew' represents the maximum size of 'new'. If an illegal tagged pattern (section) has been specified in 'sub', the error message
```
                  ? In CatSub: illegal section.
```
    is displayed to the user on Error Output.

SEE ALSO
    getpat(3), makpat(3), amatch(3), getsub(3), maksub(3)

DIAGNOSTICS
    If an illegal section is specified, a comment to that effect is displayed on Error Output.

NAME

Chcopy - copy character into buffer, increment pointer, EOS terminate

SYNOPSIS

```
subroutine chcopy(c, buf, i)

character c, buf(ARB)
integer i
```

DESCRIPTION

'chcopy' copies 'c' into 'buf(i)', increments 'i', and places an EOS after 'c' in 'buf'. This routine assumes that there is enough room in 'buf' for BOTH the character and the EOS.

SEE ALSO

addset(3), stcopy(3), scopy(3)

DIAGNOSTICS

none

NAME
    Clower - fold c to lower case

SYNOPSIS
```
    c = clower(c)

    character c
```

DESCRIPTION
    Fold character c to lower case, if not already there. If c is not alphabetic, returns it unchanged.

SEE ALSO
    fold(3), upper(3), clower(3)

DIAGNOSTICS
    None

NAME

Concat - concatenate 2 strings together

SYNOPSIS

```
call concat(buf1, buf2, outstr)

character buf1(ARB), buf2(ARB), outstr(ARB)
```

DESCRIPTION

Copies the arrays buf1 and buf2 into the array outstr.

All arrays are ascii character arrays stored one character per array element.

It is perfectly legal for ‘buf1’ and ‘outstr’ to be the same arrays, which results in ‘buf2’ being appended to ‘buf1’.

SEE ALSO

scopy(3), stcopy(3), addset(3)

DIAGNOSTICS

None

NAME

    Ctoc - copy string-to-string, observing length limits

SYNOPSIS

```
    integer function ctoc (from, to, len)
    integer len
    character from (ARB), to (len)
```

DESCRIPTION

'Ctoc' copies an EOS-terminated unpacked string from one array to another, observing a maximum-length constraint on the destination array. The function return is the number of characters copied (i.e., the length of the string in the parameter 'to').

Note that the other string copy routine, 'scopy', is not protected; if the length of the source string exceeds the space available in the destination string, some portion of memory will be garbled.

IMPLEMENTATION

A simple loop copies characters from 'from' to 'to' until an EOS is encountered or all the space available in the destination array is used up.

ARGUMENTS MODIFIED

    to

SEE ALSO

    scopy(3), ctoi(3)

NAME

Ctodi - convert character string to double integer array

SYNOPSIS

```
subroutine ctodi(buf, i, di)

character buf(ARB)
integer i, di(2)
```

DESCRIPTION

'ctodi' converts the numeric string starting at 'buf(i)' into a double integer array, as described in 'initdi'. The index 'i' is left at the next character after the converted numeric string.

SEE ALSO

ditoc(3), initdi(3), incrdi(3), decrdi(3), adddi(3), subdi(3)

DIAGNOSTICS

none

NAME

Ctoi - convert string at in(i) to integer, increment i

SYNOPSIS

```
n = ctoi(in, i)

character in(ARB)
integer i               # i is incremented
integer n is returned as the converted integer
```

DESCRIPTION

Ctoi converts the character string at "in(i)" into an integer. A leading minus sign ('-') is allowed. Leading blanks and tabs are ignored; any subsequent digits are converted to the correct numeric value. The first non-digit seen terminates the scan; upon return, "i" points to this position. "n" is returned as the value of the integer.

The "in" array is an ascii character array terminated with an EOS marker (or a non-numeric character).

Zero is returned if no digits are found.

SEE ALSO

itoc(3)

DIAGNOSTICS

There are no checks for machine overflow.

NAME

Cupper - convert character to upper case

SYNOPSIS

```
c = cupper(c)

character c
```

DESCRIPTION

CUPPER converts ascii character c to upper case, if not already there. Non-alphabetic characters are returned unchanged.

SEE ALSO

upper(3), clower(3), fold(3)

DIAGNOSTICS

None

NAME

Decrdi - decrement double integer array

SYNOPSIS

```
decrdi(dblint)

integer dblint(2)

expands into:

      {
      dblint(2) = dblint(2) - 1
      if (dblint(2) < 0)
        {
        dblint(1) = dblint(1) - 1
        dblint(2) = 9999
        }
      }
```

DESCRIPTION

Invocation of this macro causes the double integer argument to be decremented by one, with an appropriate carry occurring, if necessary. See the entry for ‘initdi’ for more information on the double integer construct.

SEE ALSO

initdi(3), incrdi(3), adddi(3), subdi(3)

DIAGNOSTICS

NAME

Delete - remove a symbol from a symbol table

SYNOPSIS

```
subroutine delete (symbol, table)
character symbol (ARB)
pointer table
```

DESCRIPTION

'Delete' removes the character-string symbol given as its first argument from the symbol table given as its second argument. All information associated with the symbol is lost.

The symbol table specified must have been generated by the routine 'mktabl'.

If the given symbol is not present in the symbol table, 'delete' does nothing; this condition is not considered an error.

IMPLEMENTATION

'Delete' calls 'stlu' to determine the location of the given symbol in the symbol table. If present, it is unlinked from its hash chain. The dynamic storage space allocated to the symbol's node is returned to the system by a call to 'dsfree'.

CALLS

stlu, dsfree

SEE ALSO

enter(3), lookup(3), mktabl(3), rmtabl(3), stlu(3), dsget(3), dsfree(3), dsinit(3), sctabl(3)

NAME

Disize - determine size of file as double integer array

SYNOPSIS

```
integer function disize(file, di)

character file(FILENAMESIZE)
integer di(2)
```

DESCRIPTION

'disize' opens 'file', counts the number of characters as a double integer, closes the file, and returns the value OK. If the file could not be opened, a value of ERR is returned. Consult the entry for 'initdi' for more information on double integers.

SEE ALSO

fsize(3), initdi(3)

DIAGNOSTICS

A value of ERR is returned if the file could not be opened.

NAME
    Ditoc - convert a double integer array to a character string

SYNOPSIS
```
    integer function ditoc(di, buf, size)

    integer di(2), size
    character buf(size)
```

DESCRIPTION
    'ditoc' converts the double integer in 'di' into a character string in buf. The length of the generated string is returned as the value of the function. The entry on 'initdi' can be consulted for more information on double integers.

SEE ALSO
    ctodi(3), itoc(3), initdi(3), incrdi(3), decrdi(3), adddi(3), subdi(3)

DIAGNOSTICS
    none

NAME

Dopack - pack words at TAB stops and flush line, if required

SYNOPSIS

```
subroutine dopack(word, nxtcol, rightm, buf, fd)

filedes fd
integer nxtcol, rightm
character word(ARB), buf(MAXLINE)
```

DESCRIPTION

'dopack' packs 'word' into 'buf', aligning word at the next available tab stop, which are taken to be every 16 characters. If 'buf' cannot be added to without exceeding 'rightm', 'buf' will be flushed to 'fd' and 'word' packed into 'buf' starting in column 1. At least one word is packed into 'buf', regardless of length, to assure that some progress is made in outputting the data.

SEE ALSO

inpack(3), flpack(3)

DIAGNOSTICS

none

NAME
    Dsdecl - declare storage for Dynamic Memory routines

SYNOPSIS
    DS_DECL(Mem,MEM_SIZE)

    expands into:

```
            integer Mem(MEM_SIZE)
            character cMem(arith(MEM_SIZE,*,CHAR_PER_INT))
            equivalence (Mem(1),cMem(1))

            common / cdsmem / Mem
```

DESCRIPTION
    This macro invocation must appear in the program units which invoke any of the following routines: dsinit, iminit, tbinit. This macro causes the common block which is used by the dynamic storage routines to be generated into the program with a size determined by the constant MEM_SIZE. The same value of MEM_SIZE must be used in the calls to dsinit, iminit and tbinit as is used in the DS_DECL declaration.

    The user must have defined MEM_SIZE prior to the invocation of DS_DECL, usually via a statement of the form

```
                         define(MEM_SIZE,4000)
```

    for example.

SEE ALSO
    dsinit(3), iminit(3), tbinit(3)

DIAGNOSTICS

NAME

Dsfree - free a block of dynamic storage

SYNOPSIS

```
subroutine dsfree (block)
pointer block
```

DESCRIPTION

'Dsfree' returns a block of storage allocated by 'dsget' to the available space list. The argument must be a pointer returned by 'dsget'.

See the remarks under 'dsget' for required initialization measures.

IMPLEMENTATION

'Dsfree' is an implementation of Algorithm B on page 440 of Volume 1 of The Art of Computer Programming, by Donald E. Knuth. The reader is referred to that source for detailed information.

'Dsfree' and 'dsget' maintain a list of free storage blocks, ordered by address. 'Dsfree' searches the list to find the proper location for the block being returned, and inserts the block into the list at that location. If blocks on either side of the newly-returned block are available, they are coalesced with the new block. If the block address does not correspond to the address of any allocated block, 'dsfree' remarks "attempt to free unallocated block" and returns to the user.

BUGS/DEFICIENCIES

The algorithm itself is not the best.

SEE ALSO

dsget(3), dsinit(3)

NAME
    Dsget - obtain a block of dynamic storage

SYNOPSIS
```
    pointer function dsget (w)
    integer w
```

DESCRIPTION
    'Dsget' searches its available memory list for a block that is at least as large as its first argument. If such a block is found, its index in the memory list is returned; otherwise, the constant LAMBDA is returned.

    In order to use 'dsget', the following declaration must be present:
```
DS_DECL (mem, MEMSIZE)
```
    where MEMSIZE is supplied by the user, and may take on any positive value between 6 and 32767, inclusive. Furthermore, memory must have been initialized with a call to 'dsinit':
```
call dsinit (MEMSIZE)
```

IMPLEMENTATION
    'Dsget' is an implementation of Algorithm A' on pages 437-438 of Volume 1 of The Art of Computer Programming, by Donald E. Knuth. The reader is referred to that source for detailed information.

    'Dsget' searches a linear list of available blocks for one of sufficient size. If none are available, a value of LAMBDA is returned; otherwise, the block found is broken into two pieces, and the index (in array 'mem') of the piece of the desired size is returned to the user. The remaining piece is left on the available space list. Should this procedure cause a block to be left on the available space list that is smaller than a threshhold size, the few extra words are awarded to the user and the block is removed entirely, thus speeding up the next search for space.

BUGS/DEFICIENCIES
    It is somewhat annoying for the user to have to declare the storage area, but Fortran prevents effective use of pointers, so this inconvenience is necessary for now.

SEE ALSO
    dsfree(3), dsinit(3), dsdecl(3)

NAME

Dsinit - initialize dynamic storage space

SYNOPSIS

```
subroutine dsinit (w)
integer w
```

DESCRIPTION

'Dsinit' initializes an area of storage in the common block CDSMEM so that the routines 'dsget' and 'dsfree' can be used for dynamic storage allocation. The memory to be managed must be supplied by the user, by a declaration of the form:

```
DS_DECL (mem, MEMSIZE)
```

The memory size must be passed to 'dsinit' as its argument:

```
call dsinit (MEMSIZE)
```

IMPLEMENTATION

'Dsinit' sets up an available space list consisting of two blocks, the first empty and the second containing all remaining memory. The first word of memory (below the available space list) is set to the total size of memory; this information is used only by the dump routines 'dsdump' and 'dsdbiu'.

CALLS

error

SEE ALSO

dsget(3), dsfree(3), dsdecl(3)

NAME
Dstime - determine if the date is daylight savings time

SYNOPSIS
```
integer function dstime(date)

integer date(7)
```

DESCRIPTION
'Dstime' determines whether the given date (in the format as returned by a 'getnow' call) corresponds to daylight savings time. If this is true, a value of YES is returned, otherwise, NO.

IMPLEMENTATION
If the month specified is > 4 (April) and < 10 (October), then YES. If the month specified is < 4 or > 10, then NO. If the month = 4, and the day is < the last Sunday, then NO, otherwise, YES. If the month = 10, and the day is < the last Sunday, then YES, otherwise, NO.

CALLS
wkday(3)

SEE ALSO
getnow(2), wkday(3)

NAME

Entdef - enter a new symbol definition, discarding any old one

SYNOPSIS

```
subroutine entdef(name, defn, table)

character name(ARB), defn(ARB)
pointer table
```

DESCRIPTION

'entdef' enters a (name,defn) pair into the symbol table 'table'. If any old definitions for 'name' exist, they are purged. 'table' must have been obtained by a call to 'mktabl'. If the (name,defn) pair cannot be stored in the table, the error message

```
in entdef: no room for new definition.
```

is displayed on error output.

SEE ALSO

mktabl(3), ludef(3)

DIAGNOSTICS

If the symbol definition cannot be entered, an error message is displayed to the user.

NAME
Enter - place symbol in symbol table

SYNOPSIS
```
integer function enter (symbol, info, table)
character symbol (ARB)
integer info (ARB)
pointer table
```

DESCRIPTION
'Enter' places the character-string symbol given as its first argument, along with the information given in its second argument, into the symbol table given as its third argument. If the symbol is successfully entered in the table, the value of OK is returned; otherwise, the value ERR is returned.

The symbol table used must have been created by the routine 'mktabl'. The size of the info array must be at least as large as the symbol table node size, determined at table creation time.

Should the given symbol already be present in the symbol table, its information field will simply be overwritten with the new information.

'Enter' uses the dynamic storage management routines, which require initialization by the user; see 'dsinit' for further details.

IMPLEMENTATION
'Enter' calls 'stlu' to find the proper location for the symbol. If the symbol is not present in the table, a call to 'dsget' fetches a block of memory of sufficient size, which is then linked onto the proper chain from the hash table. Once the location of the node for the given symbol is known, the contents of the information array are copied into the node's information field.

CALLS
stlu, dsget

SEE ALSO
lookup(3), delete(3), mktabl(3), rmtabl(3), stlu(3), dsget(3), dsfree(3), dsinit(3), sctabl(3)

NAME
    Equal - compare str1 to str2; return YES if equal

SYNOPSIS
```
    stat = equal(str1, str2)

    character str1(ARB), str2(ARB)
    integer stat is returned as YES/NO
```

DESCRIPTION
    Compares two strings, returning YES if they are the same, NO if they differ. Each string is an ascii character array terminated with an EOS marker.

SEE ALSO
    strcmp(3)

DIAGNOSTICS
    None

NAME

Error - print single-line message and terminate execution

SYNOPSIS

```
call error (message)

integer message          #message is a hollerith array
```

DESCRIPTION

Error writes the message onto the standard error file ERROUT. A NEWLINE is always generated, even though one may not appear in the message. Endst is called and execution ceases.

Error is essentially a call to 'remark' and then to 'endst'.

The message array is a Fortran hollerith string in the format generated by the Ratfor quoted string capability. On some systems, it may be necessary to terminate the string with a '.' or other end-of-string marker.

SEE ALSO

remark(2), putlin(2), prompt(2), endst(2)

DIAGNOSTICS

None

NAME

Esc - map array(i) into escaped character, if appropriate

SYNOPSIS

```
character function esc(array, i)

character array(ARB)
integer i              # i will be incremented
```

DESCRIPTION

This function checks array(i) for the existence of an escape character (as defined by ESCAPE in the general symbol definitions). If an escape is found and is appropriate, array(i+1) is returned as the escaped character. If no escape is found, the character 'array(i)' is returned.

Those characters which have special meaning are:

b	backspace	(BS)	^H
f	formfeed	(FF)	^L
l	linefeed	(LF)	^J
n	newline	(LF)	^J
r	return	(CR)	^M
t	tab	(HT)	^I

In addition, specifying '@ddd', where '0' <= d <= '7', results in the encoding of a character with that octal representation. Therefore, a ^Z character (SUB or 8%026) could be specified as '@026'.

If the character after the escape is not one of the above or a string of digits, then that character is returned, unchanged.

SEE ALSO

index(3), type(3)

DIAGNOSTICS

None

NAME

Exppth - generate pointers to the path fields in a filename

SYNOPSIS

```
subroutine exppth(path, depth, ptr, buf)

character path(FILENAMESIZE), buf(FILENAMESIZE)
integer depth, ptr(MAXDIRECTS)
```

DESCRIPTION

Given a filename in path format in the array 'path', 'exppth' scans the pathname, filling in pointers to each path field in 'ptr', and returns the number of path fields found in 'depth'.

EXAMPLES

```
      integer depth, ptr(MAXDIRECTS)
      character scr(FILENAMESIZE)

      string path "~bin/symbols"

      call exppth(path, depth, ptr, scr)
```

Upon return from exppth, ptr(1) is 1, ptr(2) is 5, and depth is 2. The calling program can now access the individual path fields via invocations of the following form:

```
      i = ptr(2)
      junk = gtftok(path, i, scr)
```

The second path field ("symbols") is now in 'scr', awaiting further processing.

SEE ALSO

gtftok(3)

DIAGNOSTICS

none

NAME

Fcopy - copy file in to file out

SYNOPSIS

```
call fcopy (in, out)

integer in, out
```

DESCRIPTION

Assuming that both files are opened, positioned, and ready to go, the routine copies lines from the current file position until an EOF is reached on file 'in'. 'in' and 'out' are file identifiers returned by open or create.

IMPLEMENTATION

'Fcopy' simply makes repeated calls to getlin and putlin.

SEE ALSO

open(2), create(2), getlin(2), putlin(2)

DIAGNOSTICS

None

NAME

Flpack - flush any packed words

SYNOPSIS

```
    subroutine flpack(nxtcol, rightm, buf, fd)

    filedes fd
    integer nxtcol, rightm
    character buf(MAXLINE)
```

DESCRIPTION

'flpack' writes 'buf' to 'fd' if there is any data packed into 'buf', and resets nxtcol to 1.

SEE ALSO

inpack(3), dopack(3)

DIAGNOSTICS

none

NAME
    Fmtdat - convert date information to character string

SYNOPSIS
```
    subroutine fmtdat (date, time, now, form)
    character date (10), time (9)
    integer now (7), form
```

DESCRIPTION
    'Fmtdat' is used to convert date information (such as that provided by 'getnow') into human-readable graphics. The first argument is a character string to receive the representation of the current date. The second argument is a character string to receive the representation of the current time. The third argument is a date specification in the same 7-word integer array format as is returned by 'getnow' (year including century, month, day, hour, minute, second, millisecond). The fourth argument selects the format of the character representations; if form == LETTER, the date is formatted as dd-Mmm-yy; if form == DIGIT, `date' is formatted as mm/dd/yy. `time' is formatted as hh:mm:ss.

IMPLEMENTATION
    Simple integer-to-character conversions.

ARGUMENTS MODIFIED
    date, time

SEE ALSO
    getnow(2), date(1)

NAME

Fold - convert string to lower case

SYNOPSIS

```
call fold (str)

character str(ARB)
```

DESCRIPTION

Converts the array 'str' to lower case characters. Non-alphabetic characters are left unchanged. The 'str' array is ascii characters terminated by an EOS marker.

SEE ALSO

clower(3), cupper(3), upper(3)

DIAGNOSTICS

None

NAME
    Fsize - determine size of file in characters

SYNOPSIS
```
    integer function fsize(file)

    character file(FILENAMESIZE)
```

DESCRIPTION
    'fsize' opens the file, counts the number of characters using 'getch', and closes the file, returning the number of characters found as an integer. Caution must be exercised on 16-bit machines, as any files containing more than 32767 characters will not be accounted for correctly. It is probably better to use 'disize' as a rule, since the 16-bit limit will only affect files with more than 327,679,999 characters.

SEE ALSO
    disize(3)

DIAGNOSTICS
    Returns ERR if the file cannot be opened.

NAME
    Fskip - skip n characters on open file

SYNOPSIS
```
    subroutine fskip(fd, n)

    filedes fd
    integer n
```

DESCRIPTION
    'n' characters are skipped on the file open on unit 'fd'.

SEE ALSO
    acopy(3)

DIAGNOSTICS
    If an EOF is encountered before the number of characters has been skipped, the routine simply returns.

NAME

Getc - read character from standard input

SYNOPSIS

```
c = getc (c)

character c
```

DESCRIPTION

Getc reads the next character from the standard input. The character is returned in ascii format both as the functional return and in the parameter c. If the end of a line has been encountered, NEWLINE is returned. If the end of the file has been encountered, EOF is returned.

If the input file is not ascii, characters are mapped into their corresponding ascii format.

SEE ALSO

getch(2), getlin(2)

DIAGNOSTICS

None

NAME

Getpat - prepare regular expression for pattern matching

SYNOPSIS

```
integer function getpat(arg, pat)

character arg(ARB)
integer   pat(MAXPAT)
```

DESCRIPTION

Getpat is used to translate a regular expression into a format convenient for subsequent pattern matching via 'match' or 'amatch'. (For a complete description of regular expressions, see the writeup on the editor.)

A typical scenario for pattern-matching might be:

```
     stat = getpat(pattern_you_want_located, pattern_array)
     YES/NO = match(input_line, pattern_array)
```

The pattern array should be dimensioned at least MAXPAT integers long, a definition available in the standard symbol definitions file.

If the pattern can be made, the functions returns the number of integers in "pat"; otherwise it returns ERR.

Getpat is essentially a call to makpat with the following parameters:

```
                  getpat = makpat (arg, 1, EOS, pat)
```

SEE ALSO

makpat(3), match(3), amatch(3)

DIAGNOSTICS

A value of ERR is returned if a failure occurs in the encoding.

NAME
    Getsub - generate substitution pattern

SYNOPSIS
```
    integer function getsub(arg, sub)

    character arg(ARB), sub(MAXPAT)
```

DESCRIPTION
    This routine is simply a special version of 'maksub', and is equivalent to

```
               getsub = maksub(arg, 1, EOS, sub)
```

    Consult the entry for 'maksub' for what these routines do.

SEE ALSO
    maksub(3)

DIAGNOSTICS
    If an error occurs in the encoding, a value of ERR is returned.

NAME

    Getwrd - get non-blank word from in(i) into out, increment i

SYNOPSIS

```
    size = getwrd(in, i, out)

    character in(ARB), out(ARB)
    integer i               # i is incremented
    integer size is returned as the length of the word found
```

DESCRIPTION

Starting at position 'i' in array 'in', skips any leading blanks and tabs and returns the next word and its length. A word is any series of characters terminated by a BLANK, TAB, or NEWLINE. The terminator is not returned as part of the word. 'i' is incremented to the position just past the end of the word. The word is returned in array 'out'.

Both 'in' and 'out' are ascii character arrays terminated with an EOS marker.

SEE ALSO

    skipbl(3)

DIAGNOSTICS

    None

NAME
    Gitocf - general integer to character conversion with fill characters

SYNOPSIS
```
    integer function gitocf(int, str, size, base, width, fc)

    integer int, size, base, width
    character str(size), fc
```

DESCRIPTION
    'gitocf' does general formatting of integers to characters in any base and will right justify the number in a field of a given width, padding with the specified fill character. If the base specified is less than 2 or greater than 36, a base of 10 is used. If the resulting string would overflow the size of str, only the rightmost 'size-1' characters are returned. The number of characters in the string is returned as the value of the function. If 'width' is specified as 0, then no padding is performed.

SEE ALSO
    itoc(3)

DIAGNOSTICS
    none

NAME

Gtftok - fetch next path token into buffer, incrementing pointer

SYNOPSIS

```
integer function gtftok(buf, i, token)

character buf(ARB), token(FILENAMESIZE)
integer i
```

DESCRIPTION

'gtftok' fetches the next path token starting at 'buf(i)' into the array 'token', incrementing the pointer 'i' to the character which terminated the scan. The length of the token is returned as the function value. Characters which can terminate the scan are '/', '\' and EOS. Upon entry, if 'buf(i)' == '/', it is skipped.

SEE ALSO

exppth(3)

DIAGNOSTICS

none

NAME
    Gtword - get next word, subject to size limitations

SYNOPSIS
```
    integer function gtword(in, i, out, size)

    integer i, size
    character in(ARB), out(size)
```

DESCRIPTION
    'gtword' is similar to 'getwrd', except that it will only copy 'size-1' characters into 'out'. If the next word of input is too big for the buffer, the extra characters are skipped over, leaving 'i' pointing at the character which terminated the entire word, not just the portion returned in 'out'. The length of the word returned in 'out' is returned as the value of the function.

SEE ALSO
    getwrd(3)

DIAGNOSTICS
    none

NAME

Imget - fetch next token from in-memory sort area

SYNOPSIS

```
integer function imget(table, buf)

pointer table
character buf(ARB)
```

DESCRIPTION

'imget' fetches the next token from the in-memory sort area pointed to by 'table', which was returned as the function value if an 'iminit' call. If there is another token which has not been fetched yet, it is returned in 'buf' and a value of OK is returned as the value of the function; otherwise, the value EOF is returned.

SEE ALSO

iminit(3), imput(3), imsort(3)

DIAGNOSTICS

The value EOF is returned if there are no more tokens to fetch.

NAME

Iminit - initialize in-memory sort area

SYNOPSIS

```
pointer function iminit(memsiz, avetok)

integer memsiz, avetok
```

DESCRIPTION

'iminit' initializes the dynamic storage region (via a 'dsinit' call) and allocates a block of pointers for future use by 'imget', 'imput' and 'imsort'. The pointer to this block of pointers is returned as the value of the function. The program calling 'iminit' must have made the following declaration

```
                      DS_DECL(Mem,memsiz)
```

to cause the memory area used by the dynamic storage routines to be allocated. 'avetok' is an estimate of the average length of the tokens which will be inserted into the dynamic memory via 'imput' calls.

SEE ALSO

dsinit(3), imput(3), imget(3), imsort(3)

DIAGNOSTICS

The value LAMBDA is returned if the dynamic storage area is too small.

NAME
    Impath - generate search path for known files

SYNOPSIS
```
    subroutine impath(path)

    character path(arith(FILENAMESIZE,*,3))
```

DESCRIPTION
    'impath' returns a search path for use in 'loccom' when searching for known files. The search path returned depends upon whether your system has a tree-structured file system or not. If not, the path returned corresponds to:

```
                    "@e~/@e~usr/@e~bin/@e@n"
```

    while tree-structured systems return:

```
                 "@e~/tools/@e~usr/@e~bin/@e@n"
```

    Consult the entry for 'loccom' for more information on the structure of the search path.

EXAMPLES
    The program wishes to spawn the editor for the user. The following code fragment will do the trick, searching for the editor through the standard search path as used by the shell:

```
          character image(FILENAMESIZE), pid(PIDSIZE)
          character path(arith(FILENAMESIZE,*,3))
          integer loccom, spawn

          string edst   "ed"
          string args   "ed temp.fil"
          string suffix IMAGE_SUFFIX

          call impath(path)
          if (loccom(ed, path, suffix, image) != BINARY)
            call error("? Cannot locate editor image file.")
          if (spawn(image, args, pid, WAIT) == ERR)
            call error("? Error spawning editor.")
```

SEE ALSO
    loccom(2)

DIAGNOSTICS
    none

NAME

Imput - place token into in-memory sort area

SYNOPSIS

```
integer function imput(table, buf)

pointer table
character buf(ARB)
```

DESCRIPTION

'imput' places the token passed in 'buf' into the in-memory sort area pointed to by 'table', which was returned as the function value of an 'iminit' call. If there is room for the token, a value of OK is returned as the function value; otherwise, a value of ERR is returned.

SEE ALSO

iminit(3), imget(3), imsort(3)

DIAGNOSTICS

If there is no room for the token, a value of ERR is returned.

NAME
    Imrset - reset in-memory read pointer

SYNOPSIS
```
    subroutine imrset(table)

    pointer table
```

DESCRIPTION
    'imrset' resets the in-memory read pointer, such that the next 'imget' call will start reading at the beginning of the in-memory sort area. 'table' must have been obtained by a call to 'iminit'. This function is equivalent to rewinding an input file.

SEE ALSO
    iminit(3), imget(3)

DIAGNOSTICS
    none

NAME

Imsort - sort tokens in in-memory sort area

SYNOPSIS

```
subroutine imsort(table)

pointer table
```

DESCRIPTION

'imsort' sorts the string tokens stored in the in-memory sort area pointed to by 'table', which was returned as the function value of a previous 'iminit' call. The strings are sorted according to the ASCII collating sequence, with all characters being significant. Upon completion, the tokens may be fetched via 'imget' calls in sorted order.

SEE ALSO

iminit(3), imput(3), imget(3)

DIAGNOSTICS

NAME
Imuniq - unique sorted in-memory array

SYNOPSIS
```
subroutine imuniq(table)

pointer table
```

DESCRIPTION
'imuniq' scans the in-memory array generated via 'imput' calls, and possible sorted by a call to 'imsort', eliminating adjacent duplicate lines. 'table' must have been obtained by a call to 'iminit'. This is the same function as provided by the 'uniq' utility for files.

SEE ALSO
iminit(3), imput(3), imsort(3), uniq(1)

DIAGNOSTICS
none

NAME

    Incrdi - increment double integer array

SYNOPSIS

```
    incrdi(dblint)

    integer dblint(2)

    expands into:

          {
          dblint(2) = dblint(2) + 1
          if (dblint(2) >= 10000)
            {
            dblint(1) = dblint(1) + 1
            dblint(2) = 0
            }
          }
```

DESCRIPTION

    Invocation of this macro causes the double integer argument to be incremented by one, with the appropriate carry into the high integer, if necessary. See the entry for 'initdi' for more information on the double integer structure.

SEE ALSO

    initdi(3), decrdi(3), adddi(3), subdi(3)

DIAGNOSTICS

NAME
    Index - find character c in string str

SYNOPSIS
```
    loc = index(str, c)

    character str(ARB), c
    integer loc is returned as the location is str where c was located
```

DESCRIPTION
    Returns the index of the first character in 'str' that matches 'c', or zero if 'c' isn't in the array. 'Str' is an ascii character array terminated with an EOS marker. 'c' is a single ascii character.

SEE ALSO
    match(3), getpat(3), indexs(3)

DIAGNOSTICS
    None

NAME

Indexs - return index of substring in character string

SYNOPSIS

```
integer function indexs(str, sub)

character str(ARB), sub(ARB)
```

DESCRIPTION

'indexs' scans the string 'str' for the first occurrence of the substring 'sub', and returns the index into 'str' where 'sub' starts. If the substring is not found, a value of 0 is returned. The comparison is stricly character by character, as done in 'strcmp' or 'equal'.

SEE ALSO

strcmp(3), equal(3), index(3)

DIAGNOSTICS

If the substring cannot be found, a value of 0 is returned.

NAME
    Inihlp - initialize help facility on help archive

SYNOPSIS
```
    integer function inihlp(file, ptrara, ptrsiz, unit)

    integer ptrsiz
    linepointer ptrara(ptrsiz)
    filedes unit
    character file(FILENAMESIZE)
```

DESCRIPTION
    'inihlp' opens 'file' at READ access, and notes the disk address of each archive header in the linepointer array, 'ptrara'. If the number of headers is larger than 'ptrsiz', only 'ptrsiz' addresses are noted. The ratfor unit for using 'mrkhlp' and 'puthlp' is returned in 'unit'. If the file could not be opened, ERR is returned as the function value; otherwise, OK is returned.

SEE ALSO
    mrkhlp(3), puthlp(3), note(2)

DIAGNOSTICS
    If the file cannot be opened, ERR is returned.

NAME

Initdi - initialize double integer array

SYNOPSIS

```
    initdi(dblint)

    integer dblint(2)

    expands into:

          {
          dblint(1) = 0
          dblint(2) = 0
          }
```

DESCRIPTION

This macro expansion causes the double integer array argument to be initialized for use in the other double integer macros and routines. The double integer construct is used by all utilities which have to count quantities which might be larger than a 16-bit integer (32767), which seems to be most things of counting interest.

The format of the double integers is:

* the second element of the array varies from 0 to 9999

* the first element of the array is the carry from the second element

In this manner, up to 327,679,999 units of things can be counted before 16-bit architectures overflow.

SEE ALSO

incrdi(3), decrdi(3), adddi(3), subdi(3), ctodi(3), ditoc(3)

DIAGNOSTICS

NAME

Inpack - initialize data for packing subroutines

SYNOPSIS

```
    subroutine inpack(nxtcol, rightm, buf, fd)

    filedes fd
    integer nxtcol, rightm
    character buf(MAXLINE)
```

DESCRIPTION

'inpack' initializes the parameters for packing data using 'dopack' and 'flpack'. These routines pack words into a buffer, aligned in columns starting every 16 characters, using TAB characters to achieve the spacing. 'inpack' sets 'nxtcol' to 1, and returns.

SEE ALSO

dopack(3), flpack(3)

DIAGNOSTICS

none

NAME
    Itoc - convert integer to character string

SYNOPSIS
```
    length = itoc(int, str, size)

    integer int, size
    character str(ARB)
    integer length returned as the number of characters needed
```

DESCRIPTION
    Converts an integer 'int' to characters in array 'str', which is at most 'size' characters long. 'length' is returned as the number of characters the integer took, not including the EOS marker. Characters are stored in ascii character arrays terminated with an EOS marker.

    Negative numbers are handled correctly.

SEE ALSO
    ctoi(3), putdec(3), putint(3), gitocf(3)

DIAGNOSTICS
    None

NAME
    Length - compute length of string

SYNOPSIS
```
    n = length(str)

    character str(ARB)
    integer n returned as the number of characters in str
```

DESCRIPTION
    Computes the length of a character string, excluding the EOS. The string is an ascii character array terminated with an EOS marker.

SEE ALSO

DIAGNOSTICS
    None

NAME

    Logpmt - 'prompt' with history mechanism

SYNOPSIS

```
    integer function logpmt(pstr, buf, fd)

    character pstr(ARB), buf(MAXLINE)
    filedes fd
```

DESCRIPTION

    'logpmt' is semantically the same as 'prompt', with the addition that is keeps a log of each line returned to the user, and permits the user to recall and edit lines previously entered. The writeup for 'hsh', the history shell, may be consulted for the syntax of the history manipulating commands.

SEE ALSO

    prompt(2), rawpmt(3), ledpmt(3), hsh(1)

DIAGNOSTICS

    Same as for prompt(2).

NAME

Lookup - retrieve information from a symbol table

SYNOPSIS

```
integer function lookup (symbol, info, table)
character symbol (ARB)
integer info (ARB)
pointer table
```

DESCRIPTION

'Lookup' examines the symbol table given as its third argument for the presence of the character-string symbol given as its first argument. If the symbol is not present, 'lookup' returns 'NO'. If the symbol is present, the information associated with it is copied into the information array passed as the second argument to 'lookup', and 'lookup' returns 'YES'.

The symbol table used must have been created by the routine 'mktabl'. The size of the information array must be at least as great as the symbol table node size, specified at its creation.

Note that all symbol table routines use dynamic storage space, which must have been previously initialized by a call to 'dsinit'.

IMPLEMENTATION

'Lookup' calls 'stlu' to determine the location of the symbol in the table. If 'stlu' returns NO, then the symbol is not present, and 'lookup' returns NO. Otherwise, 'lookup' copies the information field from the appropriate node of the symbol table into the information array and returns YES.

ARGUMENTS MODIFIED

info

CALLS

stlu

SEE ALSO

enter(3), delete(3), mktabl(3), rmtabl(3), stlu(3), sctabl(3), dsinit(3), dsget(3), dsfree(3)

NAME

Ludef - look up a defined symbol, returning its definition

SYNOPSIS

```
integer function ludef(name, defn, table)

character name(ARB), defn(ARB)
pointer table
```

DESCRIPTION

'ludef' looks up 'name' in the symbol table 'table', returning its definition in 'defn'. If the symbol is found, a value of YES is returned as the function value, otherwise, NO. 'defn' is assumed to be large enough to hold the definition stored. 'table' must have been obtained by a call to 'mktabl'.

SEE ALSO

mktabl(3), entdef(3)

DIAGNOSTICS

Returns a value of NO if the symbol cannot be found.

NAME
Makpat - prepare regular expression for pattern matching

SYNOPSIS
```
integer function makpat(arg, from, delim, pat)

character arg(ARB), delim
integer   from, pat(MAXPAT)
```

DESCRIPTION
Makpat is similar to getpat, but slightly more general purpose. It is used to translate a regular expression into a format convenient for subsequent pattern matching via 'match' or 'amatch'. (For a complete description of regular expressions, see the writeup on the editor.)

Makpat scans "arg" starting at location "from" and terminates the scan at the 'delim' character. The characters between arg(from) and the delimiter are then encoded into a pattern suitable for subsequent matching. The function returns an index into arg of the next character past the delimiter, or ERR if there was some problem encoding the pattern.

The pattern array should be dimensioned at least MAXPAT integers long, a definition available in the standard symbol definitions file.

SEE ALSO
getpat(3), match(3), amatch(3)

DIAGNOSTICS
A value of ERR is returned if a failure occurs in the encoding.

NAME
    Maksub - make substitution string

SYNOPSIS
```
    integer function maksub(arg, from, delim, sub)

    character arg(ARB), sub(MAXPAT)
    integer from
```

DESCRIPTION
    Starting at 'arg(from)', a substitution string is encoded into 'sub' until the 'delim' character is sensed in 'arg'. The next available character position in 'arg' is returned as the value of the function. If an error occurs in the encoding, a value of ERR is returned. This function is concerned with encoding the ditto character '&' and the tagged patterns (those of the form $1 ;.. $9). it also handles escaped characters (@c).

SEE ALSO
    getsub(3), ed(1)

DIAGNOSTICS
    A value of ERR is returned if the encoding fails for any reason.

NAME

Match - match pattern anywhere on a line

SYNOPSIS

```
integer function match (lin, pat)

character lin(ARB)
integer   pat(MAXPAT)
```

DESCRIPTION

'Match' attempts to find a match for a regular expression anywhere in a given line of text. The first argument contains the text line; the second contains the pattern to be matched. The function return is YES if the pattern was found anywhere in the line, NO otherwise.

The pattern in 'pat' is a standard Software Tools encoded regular expression. 'Pat' can be generated most conveniently by a call to the routine 'makpat'.

IMPLEMENTATION

'Match' calls 'amatch' at each position in 'lin', returning YES whenever 'amatch' indicates it found a match. If the test fails at all positions, 'match' returns NO.

CALLS

amatch(3)

BUGS/DEFICIENCIES

Not exactly blindingly fast.

SEE ALSO

amatch(3), makpat(3), maksub(3), catsub(3), find(1), ch(1), ed(1)

NAME

Mktabl - make a symbol table

SYNOPSIS

```
pointer function mktabl (nodesize)
integer nodesize
```

DESCRIPTION

'Mktabl' creates a symbol table for manipulation by the routines 'enter', 'lookup', 'delete', and 'rmtabl'. The symbol table is a general means of associating data with a symbol identified by a character string. The sole argument to 'mktabl' is the number of (integer) words of information that are to be associated with each symbol. The function return is the address of the symbol table in dynamic storage space (see 'dsinit' and 'dsget'). This value must be passed to the other symbol table routines to select the symbol table to be manipulated.

If an allocation failure occurs, the value LAMBDA is returned.

Note that dynamic storage space must be initialized by a call to 'dsinit' before using any symbol table routines.

IMPLEMENTATION

'Mktabl' calls 'dsget' to allocate space for a hash table in dynamic memory. Each entry in the hash table is the head of a linked list (with zero used as a null link) of symbol table nodes. 'Mktabl' also records the nodesize specified by the user, so 'enter' will know how much space to allocate when a new symbol is entered in the table.

CALLS

dsget

SEE ALSO

enter(3), lookup(3), delete(3), rmtabl(3), stlu(3), dsget(3), dsfree(3), dsinit(3), sctabl(3)

DIAGNOSTICS

If an allocation failure occurs, the value LAMBDA is returned.

NAME
    Mrkhlp - mark help elements matching pattern

SYNOPSIS
```
    integer function mrkhlp(unit, ptrara, key, outara)

    linepointer ptrara(ARB), outara(ARB)
    filedes unit
    character key(ARB)
```

DESCRIPTION
    'mrkhlp' goes through the set of archive modules pointed to by 'ptrara' and copies those which match the pattern specified by 'key' into 'outara', terminating the list with an element having the value NULLPOINTER. If the key is one of the strings "%" or "?", all elements in 'ptrara' are copied into 'outara'; otherwise, only the module with a name which matches 'key' exactly (via an 'equal' call) is copied. If none of the modules match 'key', ERR is returned; otherwise, OK is returned.

SEE ALSO
    inihlp(3), puthlp(3), equal(3)

DIAGNOSTICS
    If none of the modules match 'key', ERR is returned.

NAME

    Ngetch - get a (possibly pushed back) character

SYNOPSIS

```
    character function ngetch(c, fd)

    character c
    filedes fd
```

DESCRIPTION

    'ngetch' fetches the next character into the variable 'c' and also returns it as its value. If there are any characters on the push back buffer, the most recently pushed back character will be returned and removed from the buffer.

SEE ALSO

    putbak(3), pbstr(3), pbinit(3), pbdecl(3)

DIAGNOSTICS

    If an end of file is reached, EOF is returned.

NAME

Pbdecl - declare push-back buffer storage

SYNOPSIS

```
    PB_DECL(Buffer_size)

    expands into:

          integer pbp, pbsize
          character pbbuf(Buffer_size)

          common / cpback / pbp, pbsize, pbbuf
```

DESCRIPTION

Invocation of this macro causes the buffer and associated variables needed by the push-back buffer routines to be declared. This macro expansion must appear in the modules which invoke the ‘pbinit’ routine. The same value of ‘Buffer_size’ must be used in the ‘pbinit’ call that is used in the PB_DECL declaration.

‘Buffer_size’ must have been defined prior to the expansion of the macro, usually by a statement of the form:

```
                    define(Buffer_size,512)
```

for example.

SEE ALSO

pbinit(3)

DIAGNOSTICS

NAME
    Pbinit - initialize push-back buffer

SYNOPSIS
```
    subroutine pbinit(bufsiz)

    integer bufsiz
```

DESCRIPTION
    'pbinit' permits the user to initialize the push-back buffer without knowledge of its implementation. After initialization, 'ngetch', 'putbak' and 'pbstr' may be used. The following declaration must be made in the module which calls 'pbinit' to create the common block which these routines use:

```
                        PB_DECL(bufsiz)
```

    This declaration causes a character array 'bufsiz' characters to be created for use by the routines.

SEE ALSO
    ngetch(3), putbak(3), pbstr(3), pbdecl(3)

DIAGNOSTICS

NAME
Pbstr - push string onto push back buffer

SYNOPSIS
```
subroutine pbstr(in)

character in(ARB)
```

DESCRIPTION
'pbstr' pushes the characters in the string 'in' onto the push back buffer, from which they will be retrieved via future 'ngetch' calls. If there is insufficient room in the buffer for the characters, an error message to that effect is displayed and the program terminated.

SEE ALSO
pbinit(3), putbak(3), ngetch(3), pbdecl(3)

DIAGNOSTICS
If there is no room for the string, an error message is displayed and the program is terminated.

NAME

Putbak - push character onto push back buffer

SYNOPSIS

```
subroutine putbak(c)

character c
```

DESCRIPTION

'putbak' pushes 'c' onto the push back buffer, from which it will be removed via a future 'ngetch' call. If there is no room for the character, an error message will be displayed to that effect and the program terminated.

SEE ALSO

pbinit(3), pbstr(3), ngetch(3), pbdecl(3)

DIAGNOSTICS

If there is no room for the character, an error message is displayed and the program terminated.

NAME

Putc - write character to standard output

SYNOPSIS

```
call putc (c)

character c
```

DESCRIPTION

Putc writes a character onto the standard output file (STDOUT). If c is a NEWLINE character, the appropriate action is taken to indicate the end of the record on the file. The character is assumed to be in ascii format; however, if the output file is not ascii, characters are mapped into their corresponding format.

SEE ALSO

putch(2), putlin(2)

DIAGNOSTICS

None

NAME
    Putdec - write integer n in field width >=w

SYNOPSIS
    call putdec(n, w)

    integer n, w

DESCRIPTION
    This routine writes onto the standard output the number 'n' as a string of at least 'w' characters, including a sign if 'n' is negative. If fewer than 'w' characters are needed, blanks are inserted to the left to make up the count; if more than 'w' are needed, more are provided.

SEE ALSO
    itoc(3), putint(3)

DIAGNOSTICS
    None

NAME
Puthlp - output marked modules from help archive

SYNOPSIS
```
    subroutine puthlp(unit, outara, key, out, putout)

    linepointer outara(ARB)
    filedes unit, out
    character key(ARB)
    external putout
```

DESCRIPTION
'puthlp' outputs the help archive entries marked in 'outara' onto ratfor unit 'out' using the external routine 'putout' via calls of the form

```
                     call putout(buf, out)
```

in a format depending upon 'key'. If 'key' is the string "%", only the first line of each marked entry is output; otherwise, the second through n-th lines of each entry is output.

SEE ALSO
inihlp(3), mrkhlp(3)

DIAGNOSTICS

NAME

Putint - write integer n onto file fd in field width >=w

SYNOPSIS

```
call putint( n, w, fd)

integer n, w, fd
```

DESCRIPTION

This routine writes on the file specified by 'fd' the number 'n' as a string of at least 'w' characters, including a sign if 'n' is negative. If fewer than 'w' characters are needed, blanks are inserted to the left to make up the count; if more than 'w' are needed, more are provided. If 'w' is negative, the number is left-justified in the field.

'Fd' is a a file descriptor as returned by open or create.

SEE ALSO

itoc(3), putdec(3)

DIAGNOSTICS

None

NAME
    Putlnl - output line and flush, if necessary

SYNOPSIS
```
    subroutine putlnl(buf, fd)

    character buf(ARB)
    filedes fd
```

DESCRIPTION
    'putlnl' calls 'putlin' to output the line. It then checks to see if the last character in the buffer is a NEWLINE ('@n'); if not, it outputs a NEWLINE character to flush the line. If 'buf' is empty, a NEWLINE character is output.

SEE ALSO
    putlin(2)

DIAGNOSTICS
    None

NAME
    Putptr - output linepointer as a character string

SYNOPSIS
```
    subroutine putptr(ptr, fd)

    linepointer ptr
    filedes fd
```

DESCRIPTION
    'putptr' formats the linepointer 'ptr' using 'ptrtoc', and outputs the resulting string to the ratfor unit 'fd'.

SEE ALSO
    ptrtoc(2), note(2), seek(2)

DIAGNOSTICS
    none

NAME

Putstr - write str onto file fd in field width >=w

SYNOPSIS

```
call putstr( str, w, fd)

character str(ARB)
integer w, fd
```

DESCRIPTION

Putstr writes the character string 'str' onto the file specified by 'fd', in a field at least 'w' characters long. If fewer than 'w' characters are needed, blanks are inserted to the left to make up the count; if more than 'w' are needed, more are provided. If 'w' is negative, the characters are left-justified in the field.

'Fd' is a a file descriptor as returned by open or create.

SEE ALSO

putch(2), putlin(2), remark(2), error(3)

DIAGNOSTICS

None

NAME
    Query - print command usage information on request

SYNOPSIS
```
    subroutine query (usage)
    hollerith_string usage (ARB)
```

DESCRIPTION
    Many Software Tools commands will supply usage information when invoked with a single argument consisting only of a question mark. 'Query' exists to simplify this convention for the programmer.

    The sole argument is a period-terminated hollerith literal, such as that passed to 'error'.

    When called, 'query' checks to see that the command calling it was invoked with exactly one argument, and that that argument is a question mark. If so, the usage message is passed along to 'error' and the command terminates. If not, 'query' returns quietly.

IMPLEMENTATION
    Two calls to 'getarg', some tests, and a call to 'error'.

CALLS
    error

SEE ALSO
    error(3)

NAME

Rmdef - remove a symbol and its definition from a symbol table

SYNOPSIS

```
    subroutine rmdef(symbol, table)

    character symbol(ARB)
    pointer table
```

DESCRIPTION

'rmdef' removes a symbol and its definition from the symbol table 'table'. 'table' must have been obtained by a call to 'mktabl'.

SEE ALSO

mktabl(3), ludef(3), entdef(3)

DIAGNOSTICS

NAME
    Rmtabl - remove a symbol table

SYNOPSIS
```
    subroutine rmtabl (table)
    pointer table
```

DESCRIPTION
    'Rmtabl' is used to remove a symbol table created by 'mktabl'. The sole argument is the address of a symbol table in dynamic storage space, as returned by 'mktabl'.

    'Rmtabl' deletes each symbol still in the symbol table, so it is normally not necessary to empty a symbol table before deleting it. However, if the information associated with a symbol includes a pointer to dynamic storage space, the space will not be reclaimed. (This problem can be averted by scanning the symbol table with 'sctabl' and freeing dynamic objects, then removing the symbol table with 'rmtabl'.)

    Please see the manual entry for 'dsinit' for instructions on initializing the dynamic storage space used by the symbol table routines.

IMPLEMENTATION
    'Rmtabl' traverses each chain headed by the hash table created by 'mktabl'. Each symbol table node encountered along the way is returned to free storage by a call to 'dsfree'. Once all symbols are removed, the hash table itself is returned by a similar call.

CALLS
    dsfree

SEE ALSO
    mktabl(3), enter(3), lookup(3), delete(3), dsget(3), dsfree(3), dsinit(3), sctabl(3)

NAME

Scopy - copy string at from(i) to to(j)

SYNOPSIS

```
call scopy(from, i, to, j)

character from(ARB), to(ARB)
integer i, j
```

DESCRIPTION

Copies the (sub)string of 'from', starting in location 'i', into array 'to', starting at 'j'.

SEE ALSO

stcopy(3), addset(3), concat(3)

DIAGNOSTICS

None

NAME
    Sctabl - scan all symbols in a symbol table

SYNOPSIS
```
    integer function sctabl (table, symbol, info, posn)
    pointer table, posn
    integer info (ARB)
    character symbol (ARB)
```

DESCRIPTION
    'Sctabl' provides a means of accessing all symbols present in a symbol table (c.f. 'mktabl') without knowledge of the table's internal structure. After a simple initialization (see below), successive calls to 'sctabl' return symbols and their associated information. When the return value of 'sctabl' is EOF, the entire table has been scanned.

    The first argument is the index in dynamic storage of the symbol table to be accessed. (This should be the value returned by 'mktabl'.)

    The second and third arguments receive the character text of and integer information associated with the symbol currently under scan.

    The fourth argument is used to keep track of the current position in the symbol table. It must be initialized to zero before 'sctabl' is called for the first time for a given scan.

    The function return is EOF when the entire table has been scanned, not EOF otherwise.

IMPLEMENTATION
    If 'posn' is zero, 'sctabl' assigns the location of a two-word block in the table header to it. These words are used to keep track of (1) the hash table bucket currently in use and (2) the position in the bucket's list of the next symbol. If a symbol is available in the current list, 'sctabl' returns its data and records the position of the next symbol in the list; otherwise, it moves to the next bucket and examines that list. If there are no more buckets in the symbol table, EOF is returned as the function value and 'posn' is set to zero.

ARGUMENTS MODIFIED
    symbol, info, posn

CALLS
    dsget, dsfree

BUGS/DEFICIENCIES

A call to 'enter' must be made to update the information associated with a symbol. If new symbols are entered or old symbols deleted during a scan, the results are unpredictable. The argument order is bogus; all the other symbol table routines have the table pointer as the last argument.

SEE ALSO
lookup(3), delete(3), mktabl(3), rmtabl(3), stlu(3), dsget(3), dsfree(3), dsinit(3)

NAME
    Sdrop - drop characters from a string (APL-style)

SYNOPSIS
```
    integer function sdrop (from, to, length)
    character from (ARB), to (ARB)
    integer length
```

DESCRIPTION
    'Sdrop' copies all but 'length' characters from the 'from' string into the 'to' string and returns as its result the number of characters copied. If 'length' is positive, the omitted characters are relative to the beginning of the 'from' string; if it is negative, they are relative to the end of the string.

ARGUMENTS MODIFIED
    to

CALLS
    ctoc, length

SEE ALSO
    stake(3), index(3)

NAME

    Sdupl - duplicate a string in dynamic storage

SYNOPSIS

```
    pointer function sdupl(str)

    character str(ARB)
```

DESCRIPTION

    'sdupl' allocates space for 'str' in dynamic storage, and copies the string into the allocated space. A pointer to the dynamic space is returned as the value of the function. If the allocation fails, a value of LAMBDA is returned. 'dsinit' must have been called before this function can be called.

SEE ALSO

    dsinit(3)

DIAGNOSTICS

    Returns a value of LAMBDA if the allocation fails.

NAME

Settab - set tab stops

SYNOPSIS

```
    subroutine settab(buf, tabs)

    character buf(ARB)
    integer tabs(MAXLINE)
```

DESCRIPTION

'settab' reads the token found in 'buf', and generates the tab stops in the array tabs. If 'buf' is empty, tabstops are set starting in column 9 and every 8 columns thereafter. Consult the entries for 'entab' and 'detab' for the actual arguments which can be passed in 'buf'. After this call, 'tabs' is ready for use in calling the 'tabpos' routine.

SEE ALSO

argtab(3), tabpos(3), entab(1), detab(1)

DIAGNOSTICS

none

NAME
    Shell - shell sort integer array

SYNOPSIS
```
    subroutine shell(v, n)

    integer n, v(n)
```

DESCRIPTION
    'shell' performs a shell sort on the array of integers found in v(1) ... v(n). This algorithm is to be preferred over that used in 'bubble'.

SEE ALSO
    bubble(3)

DIAGNOSTICS
    none

NAME
    Skipbl - skip blanks and tabs at str(i)

SYNOPSIS

```
    call skipbl(str, i)

    character str(ARB)
    integer i              # i is incremented
```

DESCRIPTION
    Starting at position 'i' of array 'str', increments i while str(i) is a BLANK or TAB. 'Str' is an ascii character array terminated with an EOS marker.

SEE ALSO
    getwrd(3)

DIAGNOSTICS
    None

NAME
Stake - take characters from a string (APL-style)

SYNOPSIS
```
    integer function stake (from, to, length)
    character from (ARB), to (ARB)
    integer length
```

DESCRIPTION
'Stake' copies the number of characters specified by 'length' from the 'from' string into the 'to' string and returns as its result the number of characters copied. If 'length' is positive, the characters are copied from the beginning of 'from'; if it is negative, they are copied from the end of 'from'.

ARGUMENTS MODIFIED
to

CALLS
ctoc, length

SEE ALSO
sdrop(3), index(3)

NAME
Stcopy - copy string at from(i) to to(j); increment j

SYNOPSIS
```
call stcopy(from, i, to, j)

character from(ARB), to(ARB)
integer i
integer j             # j is incremented
```

DESCRIPTION
Copies the (sub)string of 'from', starting in location 'i', into array 'to', starting at 'j'. 'j' is incremented to point to the next available position in 'to' (i.e. the EOS marker inserted by the copy). In all other respects, 'stcopy' is similar to 'scopy'.

SEE ALSO
scopy(3), concat(3), addset(3)

DIAGNOSTICS
None

NAME

    Stlu - symbol table lookup primitive

SYNOPSIS

```
    integer function stlu(symbol, node, pred, table)

    character symbol(ARB)
    pointer node, pred, table
```

DESCRIPTION

    'stlu' looks up the token 'symbol' in the symbol table 'table', returning a pointer to the symbol in 'node' if it found. The variable 'pred' is used as a scratch pointer during the search. If the symbol is found, a value of YES is returned, otherwise, NO. 'table' is the return value of 'mktabl', and the symbol would have been entered by using the 'enter' function.

SEE ALSO

    mktabl(3), enter(3)

DIAGNOSTICS

    A value of NO is returned if the symbol cannot be found in the table.

NAME
    Strcmp - compare 2 strings

SYNOPSIS
```
    stat = strcmp (str1, str2)

    character str1(ARB), str2(ARB)
    integer stat is returned as -1, 0, or +1
```

DESCRIPTION
    Strcmp compares its aguments and returns an integer greater than, equal to, or less than 0, depending on whether str1 is lexicographically greater than, equal to, or less than str2.

SEE ALSO
    equal(3)

DIAGNOSTICS
    None

NAME
    Strcpy - copy string at "from" to "to".

SYNOPSIS
```
    call strcpy( from, to)

    character from(ARB), to(ARB)
```

DESCRIPTION
    Copies the string starting at "from" into "to".

SEE ALSO
    scopy(3), stcopy(3), addset(3), concat(3)

DIAGNOSTICS
    None

NAME
Strim - trim trailing blanks and tabs from a string

SYNOPSIS
```
integer function strim (str)
character str (ARB)
```

DESCRIPTION
'Strim' is used to trim trailing blanks and tabs from the EOS-terminated string passed as its first argument. The function return is the length of the trimmed string, excluding EOS.

IMPLEMENTATION
One pass is made through the string, and the position of the last non-blank, non-tab character remembered. When the entire string has been scanned, an EOS is planted immediately after the last non-blank.

ARGUMENTS MODIFIED
str

SEE ALSO
stake(3), sdrop(3)

NAME

Subdi - subtract double integer arrays

SYNOPSIS

```
    subdi(dbl1,dbl2)

    integer dbl1(1), dbl2(2)

    expands into:

           {
           dbl2(1) = dbl2(1) - dbl1(1)
           dbl2(2) = dbl2(2) - dbl1(2)
           if (dbl2(2) < 0)
             {
             dbl2(1) = dbl2(1) - 1
             dbl1(1) = dbl1(1) + 10000
             }
           }
```

DESCRIPTION

Invocation of this macro causes the first double integer to be subtracted from the second. If a carry is necessary, it is performed. See the entry for ‘initdi’ for more information of double integers.

SEE ALSO

initdi(3), incrdi(3), decrdi(3), adddi(3)

DIAGNOSTICS

NAME
Tabpos - determine if at a tab stop

SYNOPSIS
```
integer function tabpos(column, tabs)

integer column, tabs(MAXLINE)
```

DESCRIPTION
This function returns YES/NO depending upon whether 'column' corresponds to a tab stop or not. The array 'tabs' must have been set up via a call to 'settab' before calling 'tabpos'.

SEE ALSO
settab(3), argtab(3)

DIAGNOSTICS
none

NAME
    Tbinit - initialize simple lookup table

SYNOPSIS
```
    subroutine tbinit(size)

    integer size
```

DESCRIPTION
    'tbinit' causes a symbol table to be created for the user by calling 'mktabl' in anticipation of calling 'tbinst' and 'tblook', thus providing the same functionality as the old 'lookup' and 'instal' routines from rat4 without forcing the user to worry about the dynamic storage manipulation routines. 'size' is the size of the dynamic storage region declared in the caller via

```
                       DS_DECL(Mem,size)
```

SEE ALSO
    tbinst(3), tblook(3), dsdecl(3)

DIAGNOSTICS

NAME
    Tbinst - install (name,defn) pair in lookup table

SYNOPSIS
```
    subroutine tbinst(name, defn)

    character name(ARB), defn(ARB)
```

DESCRIPTION
    'tbinst' installs the (name,defn) pair in the lookup table initialized by a 'tbinit' call. If there is no room in the table, the message "in tbinst: no room for new definition." is displayed and control returned to the user.

SEE ALSO
    tbinit(3), tblook(3)

DIAGNOSTICS
    If there is no room for the (name,defn) pair, an error message is displayed and control returned back to the caller.

NAME

Tblook - look up name in simple lookup table

SYNOPSIS

```
integer function tblook(name, defn)

character name(ARB), defn(ARB)
```

DESCRIPTION

'tblook' looks up 'name' in the lookup table. If found, its definition is copied into 'defn' and the value YES returned as the function value; otherwise, NO is returned.

SEE ALSO

tbinit(3), tbinst(3)

DIAGNOSTICS

If the name is not in the table, a value of NO is returned.

NAME
    Tooldr - locate user-specific tool directory

SYNOPSIS
```
    subroutine tooldr(direct, dtype)

    character direct(FILENAMESIZE)
    integer dtype
```

DESCRIPTION
    'tooldr' returns the directory in which the caller's tools-specific files are kept. If 'dtype' has the value LOCAL, then the string is returned in the native operating system format; otherwise, it is returned in pathname format. It is returned as an EOS terminated string.

IMPLEMENTATION
    If the system supports Tree-structured file systems, as evidenced by the definition of TREE_STRUCT_FILE_SYS in '~bin/symbols', then the tools directory is obtained by calling 'homdir' and appending the string "tools/" to it. If the system supports a flat file system, 'homdir' is simply called. The routine is called by 'impath(3)' to build the standard search path for many of the tools.

SEE ALSO
    homdir(2), impath(3)

DIAGNOSTICS

NAME
    Type - determine type of character

SYNOPSIS
```
    t = type(c)

    character c
    character t is returned as LETTER, DIGIT, or c
```

DESCRIPTION
    This function determines whether the character 'c' is a letter, a digit, or something else; it returns LETTER, DIGIT, or the character itself.

SEE ALSO
    index(3)

DIAGNOSTICS
    None

NAME

Upper - convert string to upper case

SYNOPSIS

```
call upper(str)

character str(ARB)
```

DESCRIPTION

Converts the array 'str' to upper case, if not already there. If any characters are non-alphabetic, it leaves them unchanged. 'Str' is an ascii character array terminated with an EOS marker.

SEE ALSO

cupper(3), fold(3), clower(3)

DIAGNOSTICS

None

NAME

Wkday - get day-of-week corresponding to month, day, year

SYNOPSIS

```
integer function wkday (month, day, year)
integer month, day, year
```

DESCRIPTION

'Wkday' is used to return the day-of-the-week corresponding to a given date. The three arguments completely specify the date: the month (1-12), day (1-28, 29, 30, or 31), and year (e.g. 1980). The function return is the ordinal number of the day-of-the-week (1 == Sunday, 7 == Saturday).

IMPLEMENTATION

Zeller's Congruence.

SEE ALSO

getnow(2), fmtdat(3), date(1)

```
  ____            _   _               _  _
 / ___|  ___  ___| |_(_) ___  _ __   | || |
 \___ \ / _ \/ __| __| |/ _ \| '_ \  | || |_    _____
  ___) |  __/ (__| |_| | (_) | | | | |__   _|  |_____|
 |____/ \___|\___|\__|_|\___/|_| |_|    |_|

  ____       _
 |  _ \ _ __(_)_ __ ___   ___ _ __ ___
 | |_) | '__| | '_ ` _ \ / _ \ '__/ __|
 |  __/| |  | | | | | | |  __/ |  \__ \
 |_|   |_|  |_|_| |_| |_|\___|_|  |___/
```

NAME

Ed - text editor

A Tutorial Introduction to the Software Tools TEXT EDITOR

B. Kernighan
Bell Laboratories

and

M.J. Gralia
Johns Hopkins University - Applied Physics Laboratory

INTRODUCTION

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

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

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

DISCLAIMER

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

You must also know what character to type as the end-of-line on your particular terminal. It is almost always a "return". Throughout, we will refer to this character, whatever it is, as "newline".

CASES

And about case: it is traditional to use both upper and lower case characters when using the Software Tools, but it is not required. In describing ed, we will follow that convention, but ed will work with either.

But a caution: ed differentiates cases. If your files contain both and your terminal is in upper case, you can get into a "deadly embrace" situation in which you can see a character but can't delete it. The solution is simple - always use both upper and lower case with Software Tools.

GETTING STARTED

We'll assume that you have logged in. The easiest way to get ed is to type

```
ed      (followed by a newline)
```

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

CREATING TEXT - the Append command ``a''

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

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

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

The user tells ed what to do to his text by typing instructions called "commands". Most commands consist of a single letter. Each command is typed on a separate line. (Sometimes the command is preceded by information about what line or lines of text are to be affected - we will discuss these shortly.)

The first command is append, written as the letter

```
a
```

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

So to enter lines of text into the buffer, we just type an "a" followed by a newline, followed by the lines of text we want, like this:

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

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

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

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

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

To add more text to what we already have, just issue another "a" command, and continue typing. (Try it now - it won't always work right until we explain about line numbers.)

ERROR MESSAGES - ``?''

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

```
?
```

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

WRITING TEXT OUT AS A FILE - the Write command ``w''

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

```
w
```

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

```
w junk
```

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

```
3
```

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

LEAVING ED - the Quit command ‘‘q’’

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

```
q
```

which stands for quit. At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting.

EXERCISE 1:

Enter ed and create some text using

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

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

```
cat filename
```

Also try

```
crt filename
```

Here, you need to enter a newline (to see the next page) or "q" (to quit displaying the text).

READING TEXT FROM A FILE - the Edit command ``e''

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

```
e junk
```

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

```
3
```

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

If we use the "e" command to read a file into the buffer, then we need not use a file name after a subsequent "w" command; ed remembers the last file name used in an "e" command, and "w" will write on this file. Thus a common way to operate is

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

You can find out at any time what file named ed is remembering by typing the file command "f". In our case, if we typed

```
f
```

ed would reply

```
junk
```

READING TEXT FROM A FILE - the Read command ``r''

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

```
r junk
```

will read the file "junk" into the buffer; it adds it to the buffer (after the current line). So if we do a read after an edit:

```
e junk
r junk
```

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

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

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

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

EXERCISE 2:
Experiment with the "e" command - try reading and printing various files. You may get an error "?.", typically because you spelled the file name wrong. Try alternately reading and appending to see that they work similarly. Verify that

```
ed filename
```

is exactly equivalent to

```
ed
e filename
```

What does

```
f filename
```

do?

PRINTING THE CONTENTS OF THE BUFFER - the Print command ``p''
To print or list the contents of the buffer (or parts of it) on the terminal, we use the print command

```
p
```

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

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

Ed will respond with

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

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

```
      1,$p
```

This will print all the lines in the buffer (line 1 to the last line.)

To print the last line of the buffer, we could use

```
      $,$p
```

but ed lets us abbreviate this to

```
      $p
```

We can print any single line by typing the line number followed by a "p". Thus

```
      1p
```

produces the response

```
      Now is the time
```

which is the first line of the buffer.

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

```
      $
```

ed will print the last line of the buffer for us.

We can also use "$" in combinations like

```
      $-1,$p
```

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

EXERCISE 3:

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

```
3,1p
```

don't work.

THE CURRENT LINE - 'Dot' or '.'

Suppose our buffer still contains the six lines as above, that we have just typed

```
1,3p
```

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

```
p     (no line numbers).
```

This will print

```
to come to the aid of their party.
```

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

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

```
.     (pronounced "dot").
```

Dot is a line number in the same way that "$" is; it means exactly "the current line", or loosely, "the line we most recently did something to." We can use it in several ways - one possibility is to say

```
.,$p
```

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

Some commands change the value of dot, while others do not. The print command sets dot to the number of the last line printed; by our last command, we would have "." = "$" = 6.

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

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

This means "print the next line" and gives us a handy way to step slowly through a buffer. We can also say

```
      .-1     (or .-1p)
```

which means "print the line before the current line." This enables us to go backwards if we wish. Another useful one is something like

```
      .-3,.-1p
```

which prints the previous three lines.

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

```
      .=
```

Ed will respond by printing the value of dot.

Let's summarize some things about the "p" command and dot. Essentially "p" can be preceded by 0, 1, or 2 line numbers. If there is no line number given, it prints the "current line", the line that dot refers to. If there is one line number given (with or without the letter "p"), it prints that line (and dot is set there); and if there are two line numbers, it prints all the lines in that range (and sets dot to the last line printed.) If two line numbers are specified the first can't be bigger than the second (see Exercise 3.)

Typing a single newline will cause printing of the next line - it's equivalent to ".+1p". Try it.

DELETING LINES: the ``d'' command

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

```
      d
```

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

```
starting-line, ending-line d
```

Thus the command

```
4,$d
```

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

```
1,$p
```

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

EXERCISE 4:

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

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

```
0r filename
```

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

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

Notice that ".w" is very different from

```
.
w
```

MODIFYING TEXT: the Substitute command ``s''

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

```
s
```

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

Suppose that by a typing error, line 1 says

```
Now is th time
```

- the "e" has been left off "the". We can use "s" to fix this up as follows:

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

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

```
p
```

and get

```
Now is the time
```

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

The general way to use the substitute command is

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

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

Thus we can say

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

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

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

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

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

It's also legal to say

```
s/something//
```

which means "change 'something' to nothing," i.e., remove it. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if we had

```
Nowxx is the time
```

we can say

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

to get

```
Now is the time
```

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

EXERCISE 5:

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

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

You will get

```
on the other side of the coin
```

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

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

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

(If you get funny results using any of the characters

```
% ? $ [ *
```

read the section on "Special Characters".)

CONTEXT SEARCHING - ``/.../''

With the substitute command mastered, we can move on to another highly important idea of ed - context searching.

Suppose we have our original three line text in the buffer:

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

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

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

```
/string of characters we want to find/
```

For example, the ed line

```
/their/
```

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

```
to come to the aid of their party.
```

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

```
      ?
```

Otherwise it prints the line it found.

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

```
      /their/s/their/the/p
```

which will yield

```
      to come to the aid of the party.
```

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

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

Suppose the buffer contains the three familiar lines

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

Then the ed line numbers

```
      /Now/+1
      /good/
      /party/-1
```

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

```
      /Now/+1s/good/bad/
```

or

```
      /good/s/good/bad/
```

or

```
/party/-1s/good/bad/
```

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

```
/Now/,/party/p
```

or

```
/Now/,/Now/+2p
```

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

```
1,$p
```

but not if there were several hundred.)

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

EXERCISE 6:

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

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

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

(If you get funny results with any of the characters

```
% ? $ [ *
```

read the section on "Special Characters".)

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

```
/string/
```

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

```
//
```

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

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

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

```
\\
```

means "scan backwards for the same expression."

CHANGE and INSERT - ``c'' and ``i''

This section discusses the change command

```
c
```

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

```
i
```

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

"Change", written as

```
c
```

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

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

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

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

"Insert" is similar to append - for instance

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

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

EXERCISE 7:

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

```
start, end d
i
...text...
.
```

is almost the same as

```
start, end c
...text...
.
```

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

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

```
line-number a
...text..
.
```

appends **after** the given line, while

```
line-number i
...text...
.
```

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

BROWSING: the ``b'' command

Many times you want to look at several lines of a large file while you're using a video terminal. If you said

```
1,$p
```

the whole buffer would flash on the screen, usually too fast to read. A better way is the browse command "b". It prints just enough lines (23) to fill the CRT screen. Browse has three major forms which control what lines are displayed. "b" or "b+" prints the current line and the screen after it. "b." prints the screen centered on the current line and including it. "b-" prints the screenful before the current line.

MOVING TEXT AROUND: the ``m'' command

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

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

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

```
1,3m$
```

The general case is

```
start-line, end-line m after-this-line
```

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

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

we could reverse the two paragraphs like this:

```
/Second/,/second/m/First/-1
```

Notice the "-1" - the moved text goes after the line mentioned.

Dot gets set to the last line moved.

THE GLOBAL COMMAND ``g''

The global command "g" is used to execute an ed command on all those lines in the buffer that match some specified string. For example

```
g/peling/p
```

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

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

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

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

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

SPECIAL CHARACTERS

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

```
/x?y/
```

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

```
%   .   $   [   ]   *   @   #   !   +   {   }
```

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

```
s/@@?@*/at quest star/
```

will change "@?*" into "at quest star".

Here is a hurried synopsis of the other special characters. First, the percent "%" signifies the beginning of a line. Thus

```
/%string/
```

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

```
string
```

but not

```
the string...
```

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

```
/string$/
```

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

```
/%string$/
```

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

```
/%?$/
```

finds a line containing exactly one character.

The character "?", as we mentioned above, matches anything;

```
/x?y/
```

matches any of

```
xay
x1y
x+y
x-y
x y
x.y
```

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

```
s/?*/stuff/
```

which changes an entire line, or

```
s/?*,//
```

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

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

```
      /[1234567890]/
```

matches any single digit - any one of the characters inside the braces will cause a match.

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

```
      Now is the time
```

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

```
      s/%/(/
      s/$/)/
```

using our knowledge of "%" and "$". But the easiest way uses the "&":

```
      s/?*/(&)/
```

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

```
      s/?*/&. &!!/
```

to produce

```
      Now is the time.  Now is the time!!
```

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

```
      the end of the world
```

we could type

```
      /world/s//& is at hand/
```

to produce

```
the end of the world is at hand
```

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

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

```
s/ampersand/@&/
```

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

ACKNOWLEDGEMENT

The majority of this document has been taken, with the author's permission, from "A Tutorial Introduction to the UNIX Text Editor" by B. W. Kernighan. It has been changed only to reflect the differences between this editor and the UNIX version.

SUMMARY OF COMMANDS AND LINE NUMBERS

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

a (append) Add lines to the buffer (at line dot, unless a different line is specified). Appending continues until "." is typed on a new line. Dot is set to the last line appended.

b (browse) Display 23 lines of text, beginning at the current line. The current line will be centered if you use b. ("b dot"). Using b- will cause the previous 23 lines to be printed.

c (change) Change the specified lines to the new text which follows. The new lines are terminated by a ".". If no lines are specified, replace line dot. Dot is set to last line changed.

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

e (edit) Edit new file. Any previous contents of the buffer are thrown away, so issue a w beforehand if you want to save them.

f (file) Print remembered filename. If a name follows f the remembered name will be set to it.

g (global) g/---/command will execute the command on those lines that contain "---", which can be any context search expression.

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

m (move) Move lines specified to after the line named after m. Dot is set to the last line moved.

p (print) Print specified lines. If none specified, print line dot. A single line number is equivalent to "line-number p". A single newline prints ".+1", the next line.

q (quit) Exit from ed. Wipes out all text in buffer!!

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

s (substitute) s/string1/string2/ will substitute the characters of ‘string2’ for ‘string1’ in specified lines. If no line is specified, make substitution in line dot. Dot is set to last line in which a substitution took place, which means that if no substitution took place, dot is not changed. s changes only the first occurrence of string1 on a line; to change all of them, type a "g" after the final slash.

w (write) Write out buffer onto a file. Dot is not changed.

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

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

\---\ Context search in reverse direction. Start search at ".-1", scan to 1, wrap around to "$".

NAME

Msg - message editor

MSG Primer

Joseph Sventek
Computer Science & Mathematics Department
Lawrence Berkeley Laboratory
Berkeley, CA 94720

msg is basically a message editor. It may be used to read, write and modify files which have the message file format. There are two default files of this type in your home directory:

```
 mymail - messages sent to you by others are deposited here.
   mbox - the messages in mymail are saved here, by default.
```

msg gives the user the power to create and manage other files for conveniently sorting and categorizing messages received.

All commands to msg consist of a single character. msg then types out the rest of the command name and, if necessary, prompts for additional information needed to complete the request.

msg is entered via the following command line to the command interpreter on your machine:

```
msg [-p[n]] [filename]
```

If no filename is specified, msg defaults to the file mymail in the home directory. msg first prints out a banner identifying itself; then it reads the file specified (or mymail). If there are any messages in the file, the headers for that file are automatically displayed. Completing this, msg then prompts the user for a command character with the string

```
<-
```

The following symbols are used in the command descriptions below:

<RETURN>	the character generated by hitting the RETURN or CR key
<SPACE>	the character generated by hitting the space bar
<ESC>	the character generated by hitting the ESC key
^C	the character generated by holding down the CTRL key and hitting the key ‘C’

There are only five types of input expected by msg:

1. an msg command character
2. a message sequence specification
3. a filename
4. a confirmation character (<SPACE>)
5. an output continuation character (<SPACE>)

Whenever msg prompts for input, typing <ESC> causes the current command to be aborted, and the user is returned to command level.

The following conventions are used in the command descriptions below:

<FILE-NAME>
: This stands for any valid file specification on your system. If the tools on your system support pathname to local-name translation, any valid pathname may also be used.

<MSG-SEQUENCE>
: This input is prompted for by the string "(message sequence)". Valid responses to this prompt are:

  1. Any single message number, as listed in the headers.
  2. Any two message numbers separated by ":" or "-". This specification describes a range of message numbers (e.g. 2-5 means messages 2 and 3 and 4 and 5 in that order). If the first number is larger than the second, then the range is traversed in decreasing order. If the second message number is omitted, then the number of the last message in the current file is used.
  3. Any sequence of the previous two types separated by commas. For example,

     1,3,5-7,10

     means messages 1 and 3 and 5 through 7 and 10. <MSG-SEQUENCE> of the types described above are terminated by <RETURN>.
  4. Special types of message sequences, which are determined by the first character typed in response to the (message sequence) prompt.

character typed	action
<RETURN>	The relevant process is performed on the current message
a	The string "all messages" is displayed and the relevant action is taken on all messages

c	Identical to <RETURN>
d	The string "deleted messages" is displayed and the appropriate action is taken on those messages currently marked as deleted
f	The string "from string: " is displayed, prompting the user to supply a string to be used in a pattern match with the from fields of the headers. The characteristics of these strings are described below.
s	The string "subject string: " is displayed, prompting the user to supply a string to be used in a pattern match with the subject fields of the headers. See below for more information on the characteristics of these strings.
u	The string "undeleted messages" is displayed and the relevant action is taken on those messages currently not deleted

The strings required for the from and subject search are the same regular expressions used by the editor, find and change. Those manual entries may be consulted for more information. The string must be terminated by a <RETURN>. If a bare <RETURN> is typed in response to the string prompt, no searching is done and the command is terminated.

Whenever msg prompts for a string which must be terminated by a <RETURN> (message sequences, from or subject search strings or a filename), character editing may be performed as follows:

1. DEL(RUB) or BACKSPACE(^H) will delete the last character.
2. ^U will delete the entire string typed so far.
3. ^R will cause the current string to be retyped on the next line of the terminal. This is handy for users with hardcopy terminals, as character deletions and replacements will result in overprinted paper, and ^R can be used to see exactly what has been typed.

Since msg is a tool, both its standard input and standard output may be redirected to disk files. In particular, that is how the writeups for each individual command below was obtained, through the use of the online help facility, as well as the example dialogue described below. It should be noted that if the standard output is redirected to a file but standard input is not, none of the prompts or output of the commands typed will be seen by the user.

When operating in interactive mode, all output to the user's terminal is paged - i.e. after a screenful is displayed, the user is prompted to see if more output is desired. Positive responses to this prompt ("[type SPACE to continue]") is a <SPACE>. Any other response results in the following actions:

1. If msg was typing a large message, it will stop displaying the current message. If there are more messages to be typed in the current command, msg will then ask if the next message is desired. A negative response to this prompt ("[type SPACE for next message]") results in the discontinuation of the current command and a return to command level ("<-").
2. If the ? or h[eaders] command generates more than a screenful of lines, the user will be prompted. A negative response will result in the discontinuation of the current command.

The default page size is 22 lines. This may be modified by using the '-p[n]' switch in the arguments to msg. If n is specified, then the page size is reset to that value. Simply typing -p with no trailing number turns paging off. !!!BEWARE!!! If you turn off paging altogether, and give a command which generates a lot of output (i.e. t[ype] a[ll messages]), there is no way to stop msg until it is done. A better approach is to set n very large (say 1000 or so), so that header listings and entire messages will not be paged, but msg will stop after each message when typing multiple message sequences.

The banner that msg greets the user with is:

```
         Software Tools MSG System
             type ? for help
             type # for news
             type % for intro
```

Typing ? to the prompt results in the following information:

```
     <- ? MSG Help

     The following commands are recognized by msg:
     a[nswer]        message
     b[ackup]        to previous message and type it
     c[urrent]       message number and file
     d[elete]        message(s)
     e[xit]          and update old file
     f[orward]       message
     g[o to]         message specified and print it
     h[eaders]       print headers of message(s)
     i[nformation]   on command displayed
     j[ump]          into shell - return by typing logout to shell
     k[ey]           encryption-key  *** UNIMPLEMENTED ***
     l[ist]          message(s) in print format on file
     m[ove]          message(s) to another mail file and mark them deleted
     n[ext]          message is typed
     o[verwrite]     current file and re-read
     p[ut]           copies of message(s) in another mail file
     q[uit]          leave MSG without updating current file
     r[ead]          in another mail file
     s[ndmsg]        invoke SNDMSG to send a message (and return to MSG)
     t[ype]          message(s) on standard output
     u[ndelete]      message(s)
     #[news]         print MSG news
     ?[help]         print this list
     %[intro]        type an introduction to MSG (for first-time users)
     For more information, use the i[nformation] command.
```

Listed are the valid commands to msg. Those which are defined but unimplemented are noted as such. Expanded information for each command may be had through the use of the i[nformation] command.

Typing # results in the following display:

```
     <- # MSG News
             No news is good news!
```

As modifications are made to the system, entries will be placed in msg's database such that the news command will inform the user of recent changes.

Typing % results in the following display:

```
     <- % Introduction to MSG
     If you are a new MSG user, you probably need ONLY the following commands:

           t type message(s) on terminal; common options are 'a' for all
             messages or '<n>' (where <n> is an integer) for message <n>.

           d delete a message after reading it; common options as above.

           e exit MSG and move messages which have not been deleted to your
             mail file ('mbox' in your home directory).

           q quit MSG without updating your mail file; if there are any
             messages left, you will be notified when you next login (or
             the next time you run 'postmn').

     NOTE:   These command characters should NOT be followed by a RETURN. When
             you type one of them, MSG will immediately prompt you for more
             input.

     To print a copy of the MSG primer on the lineprinter, type

                       sh -c "msgprim | lpr"
```

This synopsis is meant for first time users, to help them in their efforts to use msg.

The following is a list of the online documentation available for each of the supported commands. The general format of the output of the i[nformation] command is:

1. A line which shows how the terminal will look when the command is used.
2. A full description of what the command does, what inputs it expects, and references to other commands with similar functionality.

```
<- information - type command character: a

Answer message number: <NUMBER>

This command causes sndmsg to be spawned as a sub-process, with the
To field being the sender of the indicated message, and the subject
field consisting of the string "Re: <SUBJECT>", where <SUBJECT> is
replaced by the subject of the indicated message.  In addition, the
message header of the answering message will contain the line

"In-reply-to: Your message of <DATE>"

where <DATE> is replaced by the date of the indicated message.
The user will be prompted for Cc addresses and the message to be sent.


<- information - type command character: b

Backing up - previous message is:

This command displays the previous message (i.e. current message - 1).
It is the inverse of the Next command.  The current message number is
decremented.  If the current message number is 1 when Backup is invoked,
an error message is displayed.


<- information - type command character: c

Current message is nn of mm messages in file <FILE-NAME>

This command displays:
  1. the number of the current message
  2. the total number of messages in the message file
  3. the file name of the currently active message file
```

<- information - type command character: d

Delete (message sequence) <MSG-SEQUENCE>

This command marks the messages specified in MSG-SEQUENCE as deleted, as indicated by an asterisk following the message number in the headers of the affected messages. The actual messages in the message file are not affected unless an Overwrite, Exit or Write command is executed before leaving MSG.

<- information - type command character: e

Exit and update old file <FILE-NAME> [type SPACE to confirm]

This command overwrites the current message file, but permits the user to leave MSG rather than re-reading the message file as Overwrite does.

<- information - type command character: f

Forward message number: <NUMBER>

This command causes sndmsg to be spawned as a sub-process, with the message consisting of the header and message body of the indicated message. The user will be prompted for To, Cc and Subject fields upon entry into sndmsg.

<- information - type command character: g

Go to message number: <NUMBER>

This command permits explicit changing of the current message number. If <NUMBER> is not in the range of acceptable values (i.e. it is less than 1 or greater than the number of messages in the file), an error message is displayed and the current message number will remain unchanged. Legal inputs for <NUMBER> are:

1. a number in the range 1 <= n <= NMSGS
2. f for the first message (message number 1)
3. l for the last message
4. <CARRIAGE-RETURN> for the current message number (a noop)

<- information - type command character: h

Headers (message sequence) <MSG-SEQUENCE>

This command displays the headers for the messages defined by the specified message sequence. Headers corresponding to deleted messages have an asterisk printed after the message number for that particular message. The format for the headers is:

```
<msg-no>  <size in characters>  <date>  <from>  <subject>
```

The headers are displayed a screenful at a time. After a screenful has been output, if there are more headers remaining to be displayed, the user is prompted with the string "[type SPACE to continue]". A response of SPACE will cause the next screenful to be displayed. Any other response terminates the listing of the headers.

```
<- information - type command character: i

Information - type command character: <COMMAND-CHARACTER>
```

This command displays full help information for those commands listed by the ? command.

```
<- information - type command character: j

Jump into shell [type SPACE to confirm]
```

This command drops the user into the Software Tools shell. All normal commands may be executed while in the shell. Control returns to MSG by typing logout to the shell.

```
<- information - type command character: l

List (message sequence) <MSG-SEQUENCE>
on file name: <FILE-NAME>
```

This command lists all the specified messages on the file specified (overwriting the current contents of <FILE-NAME>. A preface page, consisting of a FORMFEED character and the headers of the selected messages is output first, followed by each message preceded by a FORMFEED character. The file output by List can be disposed to a printer using the lpr shell command, resulting in a message on each page of the output.

```
<- information - type command character: m

Move (message sequence) <MSG-SEQUENCE>
into file name: <FILE-NAME>
```

This command is a convenient combination of the Put and Delete commands. It will first put the selected messages into the file specified and then mark the messages as deleted in the header information.

<- information - type command character: n

Next message is:

This command displays the next message (current message number + 1) and increments the current message number. If the current message is already the last one, an error message is displayed and the current message number remains unchanged.

<- information - type command character: o

Overwrite old file <FILE-NAME> [type SPACE to confirm]

This command will overwrite the current file (specified by <FILE-NAME>), eliminating any deleted messages. It then re-reads the file, re-numbering the messages.

<- information - type command character: p

Put (message sequence) <MSG-SEQUENCE>
into file name: <FILE-NAME>

This command will put the messages specified by <MSG-SEQUENCE> into the file specified by <FILE-NAME>. If the file does not exist, it will create the file and write the messages into it. If the file already exists, the messages are appended to those already in the file.

<- information - type command character: q

Quit [type SPACE to confirm]

This command allows the user to leave MSG without modifying the current message file.

<- information - type command character: r

Read file name: <FILE-NAME>

This command allows the user to use MSG on files created by previous Move or Put invocations. The current message file is closed with no modification, and the file specified is read, displaying the headers before prompting for the next command.

<- information - type command character: s

Sndmsg [type SPACE to confirm]

This command causes SNDMSG to be spawned as a sub-process, allowing the user to send a message without leaving MSG; when SNDMSG exits, MSG regains control with no changed to files, etc.

<- information - type command character: t

Type (message sequence) <MSG-SEQUENCE>

This command displays the messages specified. If more than one message is specified, the user is prompted with "[type SPACE for next message]" after each message. In addition, if a particular message is larger than one screenful, the user is prompted after each screenful. A negative response to this latter prompt results in the termination of the display of the particular message, while a negative response to the former results in termination of the Type command.

<- information - type command character: u

Undelete (message sequence) <MSG-SEQUENCE>

This command undoes the actions of the Delete command.

The following is a sample dialogue using many of the msg commands. The characters typed by the user are underlined, with the token <CR> standing for typing <RETURN>.

```
% msgtest >test.msg; msg test.msg<CR>




    Software Tools MSG System
        type ? for help
        type # for news

   1    114 25-MAR-80     Tools                 another test of mail
   2    323 27-MAR-80     Tools                 still more tests
   3    289 27-MAR-80     Tools                 testing
   4    114 01-APR-80     Tools                 test of the mail system
   5    330 03-APR-80     System                A TEST OF THE MAIL SYSTEM
   6    116 09-APR-80     Tools                 a test of the mail system
   7    308 09-APR-80     Tools                 more testing
   8     99 09-APR-80     Tools                 another test
   9    145 09-APR-80     Tools                 why doesn't mail work?
  10    129 10-APR-80     Tools                 more testing
  11    298 10-APR-80     Tools                 more testing
  12    326 10-APR-80     Tools                 Yet another test
  13    129 10-APR-80     Sventek               sventek's test
  14    314 10-APR-80     Tools                 testing again

<- headers (message sequence) 1,2,4-6<CR>
   1    114 25-MAR-80     Tools                 another test of mail
   2    323 27-MAR-80     Tools                 still more tests
   4    114 01-APR-80     Tools                 test of the mail system
   5    330 03-APR-80     System                A TEST OF THE MAIL SYSTEM
   6    116 09-APR-80     Tools                 a test of the mail system

<- type (message sequence) subject string: sventek<CR>

(message 13, 129 characters)
Date:    10-APR-80 10:43:02 - PST
From:    Sventek
Subject: sventek's test
To:      sventek, tools, system

sure hope this works

<- put (message sequence) subject string: mail<CR>
into file name: ntest.msg<CR>
<- delete (message sequence) from string: system<CR>
<- type (message sequence) deleted messages
```

```
(message 5, 330 characters)
Date: 03-APR-80 11:35:09 - PST
From: System
Subject: A TEST OF THE MAIL SYSTEM
To: allen, austin, bargmeyer, benson, gey, guest, heckman, helena,
    hogan, holmes, kreps, merrill, oracle, robinson, rtsg, scherrer,
    shoshani, sventek, sventekv, system, tabata, tape, tools

THIS IS ANOTHER TEST.  SORRY FOR THE INCONVENIENCE.


<- overwrite old file 'test.msg' [type SPACE to confirm] <SPACE>
updating...
   1    114 25-MAR-80     Tools                 another test of mail
   2    323 27-MAR-80     Tools                 still more tests
   3    289 27-MAR-80     Tools                 testing
   4    114 01-APR-80     Tools                 test of the mail system
   5    116 09-APR-80     Tools                 a test of the mail system
   6    308 09-APR-80     Tools                 more testing
   7     99 09-APR-80     Tools                 another test
   8    145 09-APR-80     Tools                 why doesn't mail work?
   9    129 10-APR-80     Tools                 more testing
  10    298 10-APR-80     Tools                 more testing
  11    326 10-APR-80     Tools                 Yet another test
  12    129 10-APR-80     Sventek               sventek's test
  13    314 10-APR-80     Tools                 testing again

<- move (message sequence) 1-5<CR>
into file name: old.msg<CR>
<- go to message number: first
(message 1, 114 characters)
Date: 25-MAR-80 12:46:23 - PST
From: Tools
Subject: another test of mail
To: tools

sure hope this works again.



<- current message is 1 of 13 messages in file 'test.msg'
<- next message is:
(message 2, 323 characters)
Date:    27-MAR-80 15:08:32 - PST
From:    Tools
Subject: still more tests
To:       allen, austin, bargmeyer, benson, gey, guest, heckman, helena,
          hogan, holmes, kreps, merrill, oracle, robinson, rtsg,
          scherrer, shoshani, sventek, sventekv, system, tabata, tape,
          tools
```

```
another test message


<- go to message number: last
(message 13, 314 characters)
Date:    10-APR-80 14:53:19 - PST
From:    Tools
Subject: testing again
To:      allen, austin, bargmeyer, benson, gey, guest, heckman, helena,
         hogan, holmes, kreps, merrill, oracle, robinson, rtsg,
         scherrer, shoshani, sventek, sventekv, system, tabata, tape,
         tools

will it never stop?

<- current message is 13 of 13 messages in file 'test.msg'
<- backing up - previous message is:
(message 12, 129 characters)
Date:    10-APR-80 10:43:02 - PST
From:    Sventek
Subject: sventek's test
To:      sventek, tools, system

sure hope this works

<- jump into shell [type SPACE to confirm] <SPACE>
% logout<CR>

<- ? MSG Help

The following commands are recognized by msg:
a[nswer]        message
b[ackup]        to previous message and type it
c[urrent]       message number and file
d[elete]        message(s)
e[xit]          and update old file
f[orward]       message
g[o to]         message specified and print it
h[eaders]       print headers of message(s)
i[nformation]   on command displayed
j[ump]          into shell - return by typing logout to shell
k[ey]           encryption-key  *** UNIMPLEMENTED ***
l[ist]          message(s) in print format on file
m[ove]          message(s) to another mail file and mark them deleted
n[ext]          message is typed
o[verwrite]     current file and re-read
p[ut]           copies of message(s) in another mail file
q[uit]          leave MSG without updating current file
r[ead]          in another mail file
```

```
s[ndmsg]        invoke SNDMSG to send a message (and return to MSG)
t[ype]          message(s) on standard output
u[ndelete]      message(s)
#[news]         print MSG news
?[help]         print this list
%[intro]        type an introduction to MSG (for first-time users)
For more information, use the i[nformation] command.


<- # MSG News
        No news is good news!

<- information - type command character: r

Read file name: <FILE-NAME>
```

This command allows the user to use MSG on files created by previous Move or Put invocations. The current message file is closed with no modification, and the file specified is read, displaying the headers before prompting for the next command.

```
<- headers (message sequence) all messages
   1*   114 25-MAR-80     Tools                 another test of mail
   2*   323 27-MAR-80     Tools                 still more tests
   3*   289 27-MAR-80     Tools                 testing
   4*   114 01-APR-80     Tools                 test of the mail system
   5*   116 09-APR-80     Tools                 a test of the mail system
   6    308 09-APR-80     Tools                 more testing
   7     99 09-APR-80     Tools                 another test
   8    145 09-APR-80     Tools                 why doesn't mail work?
   9    129 10-APR-80     Tools                 more testing
  10    298 10-APR-80     Tools                 more testing
  11    326 10-APR-80     Tools                 Yet another test
  12    129 10-APR-80     Sventek               sventek's test
  13    314 10-APR-80     Tools                 testing again

<- list (message sequence) from string: tools<CR>
on file name: tools.lst<CR>
<- undelete (message sequence) deleted messages
<- move (message sequence) from string: sventek<CR>
into file name: sventek.msg<CR>
<- current message is 12 of 13 messages in file 'test.msg'
<- headers (message sequence) undeleted messages
   1    114 25-MAR-80     Tools                 another test of mail
   2    323 27-MAR-80     Tools                 still more tests
   3    289 27-MAR-80     Tools                 testing
   4    114 01-APR-80     Tools                 test of the mail system
   5    116 09-APR-80     Tools                 a test of the mail system
   6    308 09-APR-80     Tools                 more testing
   7     99 09-APR-80     Tools                 another test
   8    145 09-APR-80     Tools                 why doesn't mail work?
   9    129 10-APR-80     Tools                 more testing
  10    298 10-APR-80     Tools                 more testing
  11    326 10-APR-80     Tools                 Yet another test
  13    314 10-APR-80     Tools                 testing again

<- overwrite old file 'test.msg' [type SPACE to confirm] <SPACE>
updating...
   1    114 25-MAR-80     Tools                 another test of mail
   2    323 27-MAR-80     Tools                 still more tests
   3    289 27-MAR-80     Tools                 testing
   4    114 01-APR-80     Tools                 test of the mail system
   5    116 09-APR-80     Tools                 a test of the mail system
   6    308 09-APR-80     Tools                 more testing
   7     99 09-APR-80     Tools                 another test
   8    145 09-APR-80     Tools                 why doesn't mail work?
   9    129 10-APR-80     Tools                 more testing
  10    298 10-APR-80     Tools                 more testing
  11    326 10-APR-80     Tools                 Yet another test
  12    314 10-APR-80     Tools                 testing again

<- read file name: ntest.msg<CR> reading...
   1    114 25-MAR-80     Tools                 another test of mail
   2    114 01-APR-80     Tools                 test of the mail system
   3    330 03-APR-80     System                A TEST OF THE MAIL SYSTEM
   4    116 09-APR-80     Tools                 a test of the mail system
   5    145 09-APR-80     Tools                 why doesn't mail work?

<- quit [type SPACE to confirm] <SPACE>
```

NAME

Ratfor - rational FORTRAN pre-processor

RATFOR PRIMER

Ratfor is a preprocessor for Fortran. Its primary purpose is to encourage readable and well-structured code while taking advantage of the universality, portability, and efficiency of Fortran. This is done by providing the control structures not available in bare Fortran, and by improving the "cosmetics" of the language.

Ratfor allows for all the features of normal Fortran, plus makes available these control structures:

"if"-"else"
"while", "for", and "repeat"-"until" for looping
"switch" for multi-way branching
"break" and "next" for controlling loop exits
statement grouping with braces

The cosmetic aspects of Ratfor have been designed to make it concise and reasonably pleasing to the eye:

free form input
unobtrusive comment convention
translation of >, <=, etc. into .GT., .LE., etc.
string data type
quoted character strings
character constants
"define" statement for symbolic constants
conditional preprocessing
"include" statement for including source files

Ratfor is implemented as a preprocessor which translates the above features into Fortran, which can then be fed into almost any Fortran compiler.

Each of the Ratfor features will now be discussed in more detail. In the following, a "statement" is any legal statement in Fortran: assignment, declaration, subroutine call, I/O, etc., or any of the Ratfor statements themselves. Any Fortran or Ratfor statement or group of these can be enclosed in braces ({}) or brackets ([]) -- to make it a compound statement, which is then equivalent to a single statement and usable anywhere a single statement can be used.

IF-ELSE

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

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

where the else part is optional. The "legal Fortran condition" is anything that can legally go into a Fortran logical IF. The Ratfor statements may be one or more valid Ratfor or Fortran statements of any kind. If more than one statement is desired, the statements must be enclosed by braces. For example,

```
          if (a > b)
              {
              k = 1
              call remark (...)
              }
          else if (a < b)
              {
              k = 2
              call remark (...)
              }
          else
              return
```

WHILE

Ratfor provides a while statement, which is simply a loop: "while some condition is true, repeat this group of statements". The syntax is

```
          while (legal Fortran condition)
              statement(s)
```

As with the if, "legal Fortran condition" is something that can go into a Fortran logical IF. The condition is tested before execution of any of the Ratfor statements, so if the condition is not met, the loop will be executed zero times. Also, as with the IF, the Ratfor statements can be any valid Ratfor or Fortran constructs. If more than one statement is desired, the statements must be enclosed by braces. For example,

```
          while (getc(c) != EOF)
              {
              c = cnvt (c)
              call putc (c)
              }
```

FOR

The "for" statement is similar to the "while" except that it allows explicit initialization and increment steps as part of the statement. The syntax is

```
          for (init; condition; increment)
              statement(s)
```

where "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 remain. A non-existent condition is treated as always true, so "for(; ;)" is an indefinite repeat. 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. Here are two examples of "for" loops:

```
          for (i=1; getarg(i, file, MAXLINE) != EOF; i=i+1)
              {
              int = open (file, READ)
              while (getlin (line, int) != EOF)
                  {
                  for (j=80; j>0; j=j-1)
                      call putc (line(j))
                  }
              call close (int)
              }
```

The above code simply reads cards from a list of files, reverses the order of the characters, and writes the cards onto a standard output file. (The "!=" means .NE.)

Groups of Fortran statements may be used in the "init" and "increment" clauses by separating the statements with commas (,). For example:

```
          for (i=1, j=1; buf(i) != EOS; i=i+2, j=j+1)
              out(j) = buf(i)
```

copies every other character in buf into consecutive locations in out.

REPEAT-UNTIL

The "repeat-until" statements allow for repetition of a group of statements until a specified condition is met. The syntax is:

```
          repeat
              statement(s)
          until (condition)
```

The "until" is optional. Once again, if more than one Ratfor statement is desired, the statements must be enclosed by brackets. If the "until" part is omitted, the result is an infinite loop which must be broken with a "break" or "next" statement (see below). An example of a repeat-until loop is:

```
          repeat
              {
              call putc (' ')
              col = col + 1
              }
          until (tabpos(col,tabs) == YES)
```

SWITCH

The "switch" statement permits the execution of multi-way branches. The syntax is:

```
          switch (expression)
              {
              case constant[,constant]*: statement(s)
              case constant[,constant]*: statement(s)
                        .
                        .
                        .
              case constant[,constant]*: statement(s)
              default:                   statement(s)
              }
```

'expression' must result in an integer or character value, which is then compared with the 'constant's enumerated in the statement block. Unlike the 'switch' statement in the C programming language, there is an implied break after each case. If more than one Ratfor statement is desired for each case, the statements must be enclosed in brackets.

It is possible to exit from a group of statements in the scope of a case label through the use of a break statement (see below). An example of the use of switch is:

```
          switch (ngetch(c, fd))
              {
              case 'a','q': x = 5
              case 'b','c': x = 10
              case EOF:     {
                            call remark("Error in input.")
                            call putbak(EOF)
                            }
              default:      x = 0
              }
```

BREAK and NEXT

Ratfor provides statements for leaving a loop early and for beginning the next iteration.

"Break" causes an immediate exit from whatever loop it is contained in (which may be a "while", "for", "repeat" or "switch"). Control resumes with the next statement after the loop. Only one loop is terminated by a "break", even if the "break" is contained inside several nested loops. For example:

```
          repeat
              {
              if (getc(c) == EOF)
                  break
              ...
              }
```

"Next" is a branch to the bottom of the loop, so it causes the next iteration to be done. "Next" goes to the condition part of a "while" or "until", to the top of an infinite "repeat" loop, and to the reinitialize part of a "for". For example:

```
          for (i=1; i<10; i=i+1)
              {
              if (array(i) == ' ')
                  next
              ...
              }
```

Breaking out of multiple loops can be achieved by specifying the number of levels to break out of after the break statement, as in:

```
          repeat
              {
              repeat
                  {
                  if (condition)
                      break 2
                  line 2
                  }
              line 1
              }
          line 0
```

Upon execution of the "break 2" statement, execution resumes at "line 0". It is probably better to use a "goto" statement when breaking out of multiple loops, since that should be a little easier to maintain and understand.

STATEMENT GROUPING AND NULL STATEMENTS

Ratfor allows a group of statements to be treated as a unit by enclosing them in braces -- { and }. This is true throughout the language: wherever a single Ratfor statement can be used, there could also be several enclosed in braces. For example:

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

If braces are not valid characters in the local operating system, the characters "[" and "]" may be used instead of "{" and "}" respectively.

Ratfor also allows for null statements, most useful after "for" and "while" statements. A semicolon alone indicates a null statement. For instance,

```
          while (getlin(line, int) != EOF)
              ;
```

would read lines from a file until the end-of-file was reached and

```
          for (i=1; line(i) == ' '; i=i+1)
              ;
```

positions after leading blanks in a line.

FREE-FORM INPUT

Statements may be placed anywhere on a line and several may appear on one line if they are separated by semicolons. No semicolon is needed at the end of each line because Ratfor assumes there is one statement per line unless told otherwise. Ratfor will, however, continue lines when it seems obvious that they are not yet done.

Any statement that begins with an all-numeric field is assumed to be a Fortran label and is placed in columns 1-5 upon output.

Statements may be passed through the Ratfor compiler unaltered by inserting a percent sign (%) as the first character on the line. The percent will be removed, the rest of the line shifted one position to the left, and the line sent out without any changes. This is a convenient way to pass regular Fortran or assembly code through the ratfor compiler.

Sequences of characters may be passed through the pre-processor unaltered by surrounding them with the tokens %(...%). This proves useful if it is necessary to interact with other system-specific software which uses RATFOR keywords or special characters. For example:

```
          call graph_label(%('X-Axis'%), %('Y-Axis'%))
```

permits the subroutine graph_label to be called with F77 character strings as the labels. Using the %(...%) construct prevents RATFOR from trying to interpret the F77 strings as character constants.

COMMENTS

A sharp character "#" in a line marks the beginning of a comment and the rest of the line is considered to be that comment. Comments and code can co-exist on the same line. For example,

```
function dummy (x)

# I made up this function to show some comments

dummy = x         #I am simply returning the parameter

return
end
```

CHARACTER TRANSLATION

Sometimes the characters >, <=, etc. are easier to read in Fortran condition statements than the standard Fortran .EQ., .LT., etc.. Ratfor allows either convention. If the special characters are used, they are translated in the following manner:

==	.EQ.
!= ^= ~=	.NE.
<	.LT.
>	.GT.
<=	.LE.
>=	.GE.
|	.OR.
&	.AND.
!	.NOT.

For example,

```
          for (i=1; i<= 5; i=i+1)
              ...

          if (j != 100)
              ...
```

STRING DATA TYPE

All character arrays in Ratfor are sequences of ASCII characters, stored right-adjusted, one per array element, with the string terminated with an EOS marker. An automatic way to initialize string characters arrays is provided. The syntax is:

```
                    string name "characters"
or
                   string name(n) "characters"
```

Ratfor will define name to be a character (or, more likely, integer) array long enough to accomodate the ASCII codes for the given character string, one per element. The last word of name is initialized to EOS. If a size is given, name is declared to be an integer array of size 'n'. If several string statements appear consecutively, the generated declarations for the array will precede the data statements that initialize them.

For example, the declarations:

```
     string errmsg "error"
     string done "bye"
```

would be converted by ratfor into the Fortran:

```
     integer error(6)
     integer done(4)
     data error(1), error(2), error(3), error(4),
     error(5), error(6) /'e', 'r', 'r', 'o', 'r', EOS/
     data done(1), done(2), done(3), done(4) /'d', 'o',
     'n', 'e', EOS/
```

The standard escape characters used in the text processing utilities (find, ch, ed, etc.) can be used inside of a string. In particular, to embed an atsign ('@') or a double quote ('"') into the string, they must be escaped, as in:

```
     string escape "Embed quote (@")"
```

QUOTED CHARACTER STRINGS

Text enclosed in matching double quotes is converted to an appropriate declaration for a ‘character’ array, and the appropriate data statements to load this array are output. The variable name will be of the form STNNNZ, where NNN is replaced by a rotating sequence number. The array will be declared long enough to place the value EOS in the last element, as for the ‘string’ declaration. Since these declarations and data statements are output immediately, the resulting FORTRAN code must be run through the program ‘ratp2’, which will reorder the code to be ANSI-66 compliant.

String literals can be continued across line boundaries by ending the line to be continued with an underline. The underline is not part of the string, nor are any leading blanks or tabs on the next line.

The normal escape sequences are permitted in quoted strings. In particular, if a quote is to be embedded in the string, it must be escaped, as in

```
"a quote (@") in a string"
```

CHARACTER LITERALS

Character constants of the form 'c' are converted to the decimal integer representation of that character in the ASCII character set. For example:

```
          call putc('!')
becomes
          call putc(33)
```

The standard escape sequences for characters (as used in find, ch and ed) are interpreted within the apostrophes. In particular, '@n' is NEWLINE, '@t' is TAB and '@@' is ATSIGN. Consult the writeup on the find utility for the complete set of escaped characters.

Note that this usage pre-empts the use of apostrophes to delimit character strings.

DEFINE

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. The syntax is:

```
                define(name, replacement string)
```

which define "name" as a macro which will be replaced with "replacement string" when encountered in the source files. As a simple example:

```
          define(ROW,10)
          define(COLUMN,25)

          dimension array (ROW, COLUMN)
```

and

```
          define(EOF,-1)
          if (getlin(line, fd) == EOF)
              ...
```

Definitions may be included anywhere in the code, as long as they appear before the defined name occurs. The names of macro may contain letters, digits, and underline characters, but must start with a letter. Upper and lower cases ARE significant (thus EOF is not the same as eof).

Any occurrences of the strings '$n' in the replacement text, where 1 <= n <= 9, will be replaced with the nth argument when the macro is actually invoked. For example:

```
               define(bump, $1 = $1 + 1)
```

will cause the source line

```
               bump(i)
```

to be expanded into

```
               i = i + 1
```

In addition to define, several other built-in macros are provided:

arith(x,op,y)	performs the "integer" arithmetic specified by op (+,-,*,/,**) on the two numeric operands and returns the result as its replacement.
incr(x)	converts the string x to a number, adds one to it, and returns the value as its replacement (as a character string).
ifelse(a,b,c,d)	compares a and b as character strings; if they are the same, c is pushed back onto the input, else d is pushed back.
substr(s,m,n)	produces the substring of s which starts at position m (with origin one), of length n. If n is omitted or too big, the rest of the string is used, while if m is out of range the result is a null string.
lentok(str)	pushes the length of the argument (# of characters) onto the input as a character string.
undefine(sym)	removes the definition for the symbol 'sym', if it is defined.

CONDITIONAL PREPROCESSING

Ratfor source code may be conditionally preprocessed, dependent upon the definition (or lack thereof) of a symbol. The syntax is

```
          ifdef(symbol)                        ifnotdef(symbol)
                .                                    .
                .                                    .
                .                                    .
          elsedef                              elsedef
                .                                    .
                .                                    .
                .                                    .
          enddef                               enddef
```

Conditionals may be nested to some maximum level (usually 10). An example of their use might be an output routine which forces the output of a characters from a string to uppercase, depending upon the definition of a symbol DO_UPPER:

```
          for (i = 1; buf(i) != EOS; i = i + 1)
              {
              ifdef (DO_UPPER)
                call putc(cupper(buf(i))
              elsedef
                call putc(buf(i))
              enddef
              }
```

INCLUDE

Files may be inserted into the input stream via the "include" command. The statement

```
          include filename
```
or
```
          include "filename"
```

inserts the file found on input file "filename" into the Ratfor input in place of the include statement. (Surrounding the filename with quotes is required if the filename contains characters other than letters, digits and underscores.) This is especially useful in inserting common blocks. For example,

```
          function exampl (x)

          include comblk

          exampl = x + z

          return
          end
```

might translate into

```
          function exampl (x)

          common /comblk/ q, r, z

          exampl = x + z

          return
          end
```

IMPLEMENTATION

Ratfor was originally written in C, a high-level language, on the Unix operating system. Our version is written in Ratfor itself, originally brought up by a bootstrap written in Fortran.

Ratfor generates code by reading input files and translating any Ratfor keywords into standard Fortran. Thus, if the first token (word) on a source line is not a keyword (like "for", "while", etc.) the entire statement is simply copied to the output with appropriate character translation and formatting. Ratfor knows very little Fortran and thus does not handle any Fortran error detection. Errors in Ratfor keyword syntax are generally noted by a message to the user's terminal along with an indication of the source line number which caused the problem.

CONCLUSIONS

Ratfor demonstrates that with modest effort Fortran-based programmers can increase their productivity by using a language that provides them with the control structures and cosmetic features essential for structured programming design. Debugging and subsequent revision times are much faster than the equivalent efforts in Fortran, mainly because the code can be easily read. Thus it becomes easier to write code that is readable, reliable, and even esthetically pleasing, as well as being portable to other environments.

EXAMPLE

The following is a sample Ratfor tool designed to show some of the commonly-used Ratfor commands. The routine reads through a list of files, counting the lines as it goes.

```
   # This is an example of a routine written in Ratfor
   # Symbols such as EOF, ERR, MAXLINE, character and filedes are
   # automatically defined (i.e. a file containing them is included)
   # by the preprocessor

   ## count - counts lines in files
   DRIVER(count)

   include comblk      # this file contains a common block which
                       # contains a variable "linect"

   character file(FILENAMESIZE), line(MAXLINE)
   integer i
   filedes fd
   integer getarg, open, getlin
   string total "total lines: "

   call query ("usage:  count file.")
   linect = 0

   # loop through the list of files

   for (i=1; getarg(i, file, FILENAMESIZE) != EOF; i=i+1)
       {
       fd = open (file, READ)          # open (attach) the file
       if (fd == ERR)                  # file could not be located
           call cant (file)
       while (getlin(line, fd) != EOF) # read and count lines
           linect = linect + 1
       call close (fd)                 # close (unattach) the file
       }

   call putlin(total, STDOUT)
   call putint (linect, 1, STDOUT)
   call putch ('@n', STDOUT)

   DRETURN
   end
```

SEE ALSO

1) Kernighan, Brian W., "Ratfor--a Preprocessor for a Rational Fortran". Software - Practice and Experience, Vol. 5, 4 (Oct-Dec 75), pp. 395-406.

2) Kernighan, Brian W. and P. J. Plauger, "Software Tools". Addison-Wesley Publishing Company, Reading, Mass., 1976.

3) The ratfor user document

4) The Unix command "rc" in the Unix Manual (RC(I))