Lawrence Berkeley National Laboratory
Recent Work

Title
SOFTWARE TOOLS PROGRAMMER'S MANUAL

Permalink
https://escholarship.org/uc/item/30g2b7b7

Author
Breckon, T.

Publication Date
1985-04-01
SOFTWARE TOOLS PROGRAMMER'S MANUAL

T. Breckon, V. Jacobson, C. Leres, V. Paxson, and N. Travis

April 1985

Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

Prepared for the U.S. Department of Energy
under Contract DE-AC03-76SF00098
DISCLAIMER

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.
LEGAL NOTICE

This book was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of the Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

Lawrence Berkeley Laboratory is an equal opportunity employer.
NAME
codeconv - RTSG coding conventions

SYNOPSIS
(see below)

DESCRIPTION

1. Introduction

The conventions specified here define the standard for all RTSG code and accompanying documentation included in the source. Although some of this document refers to the rat4 language, all source code should try to follow these standards as closely as possible.

The conventions are intended, in part, to encourage higher quality code. Every source module is required to have certain essential documentation, and the code and documentation is required to be in a format that has been found to be readable and accessible. However the conventions are also intended to provide a level of uniformity in the code produced by different programmers. Besides making the code look more professional, uniformity allows programmers to work on code written by others with less overhead in adjusting to stylistic differences. Uniformity also allows automated processing of the source. Tools can be written to generate change reports, module summaries, etc.

The conventions are divided into the following categories:

(1) Module layout
(2) Subroutine layout
(3) Code layout
(4) Constant definitions

2. Module Layout

A module is any unit of code that resides in a single source file. Thus a module may be a tool, a library of routines, or an applications task. The conventions below define the standard for the header that must come at the beginning of every source module.

The module header consists of the blocks described below. The blocks are separated by one or more blank lines and should contain no blank lines within the block (a line consisting only of "#" or ";" for example, is considered a non-blank line). All lines in the header begin with a single comment character, followed by a space. This facilitates automated processing of the header.
Title - one line containing the module name followed by a space, a dash, a space, and a short description. This line should be the same as the manual entry title line.

Revision history - each entry consists of the version number, date of revision, initials of the programmer who made the change, and a complete description of the change. See the example below for the format of the version number, date, initials, and description. The description of the change should identify each routine that was changed and the nature of the change. New entries are made at the top of the list.

General module documentation - consists of a brief description of overall module purpose and functioning.

Includes and local definitions - consists of the includes and any local definitions required by the module.

The exact format of these blocks is elaborated in the following example of a
standard module header:

```plaintext
# xxxlb - example library module
#
# Ver  Date  Who  Remarks
# --------------
# 01b  29Apr81  DNW  .xxxget, xxxput: new routines.
#                .Added check for invalid index in
#                xxxfnd.
# 01a  27Apr81  DNW  .Written
#
# This module is an example of the ratfor coding
# conventions ... 
# ...

include "xxxdef"
include "sysdef"

define(MAXSIZE,3000)
define(SCR_COMMON,
    integer scrbuf(MAXSIZE)
    common /scrcom/ scrbuf
)
```

### 3. Subroutine Layout

The following conventions define the standard layout for every subroutine and function.

Each routine is preceded by a header of documentation that consists of the following blocks. There should be no blank lines in the header but each block should be separated with a line containing only a comment character (:, #, etc).

**Title** - one line beginning with three comment characters, followed by a space, the subroutine name, a space, a dash, another space, and a short description. This line should be the same as the manual entry title line. For example:

```plaintext
### getlin - output a line onto a given file
```

**Calling sequence** - sample calling sequence, like the synopsis line of a manual entry.

**Arguments** - one line description of each argument including the function result if appropriate. The returned arguments should be identified as such. **Note:** In function and subroutine calling sequences, all passed arguments should precede all returned arguments.

**Description** - brief description of purpose and algorithm of routine.
After the header, the following declaration blocks should appear. The blocks should not contain any blank lines and are separated by a blank line.

**Routine Declaration** - declaration of subroutine or function. Note that the structure of the declaration should have no spaces before the leading parenthesis, a space before each parameter, and a space before the ending parenthesis. For example:

```fortran
integer function getlin( line, fd )
```

This structure should be adhered to in manual entry descriptions, comments in the routine header, etc. The routine declaration should be immediately followed by a NOIMPLICIT statement. NOIMPLICIT is a rat4 definition which allows the compiler to catch undeclared variables. Declarations of passed arguments should follow the NOIMPLICIT statement. Everything after the argument declarations should be indented 4 spaces (see the section on horizontal spacing).

**Local Declarations** - declarations of local variables. Variables, function and subroutine names should be kept down to six characters. Use comments rather than long variable names to describe the variable at declaration time. **Note:** All local variables should be explicitly declared. The Fortran implicit typing should not be assumed (and in fact won't work if the NOIMPLICIT statement was used).

**Common Includes/Defs** - includes of any needed common blocks or the definition if the common block was defined using a local definition. If a common block is local to the code, i.e. it is not needed by any other program or library, it should not be placed in an include file. Use a rat4 definition to define the common. For example:

```fortran
define(DS_COMMON, 
       integer dsmem(DS_MEMSIZE) 
       common /dscom/ dsmem 
)
```

The exact format of the above blocks is elaborated in the following example
of a standard routine layout.

```fortran
integer function grkget(gronk, ind, value)

 INTEGER gronk(A,RB), ind
 REAL value

 integer count    # number of elements in gronk
 integer max     # returns maximum value of args

 GRK_COMMON

 ... CODE ...

 return

 end
```

4. Code Layout

The following conventions define the standard for the graphic layout of ratfor code.

4.1. Vertical Spacing

(1) Use blank lines to make code more readable and to group logically related sections of code together.

(2) Do not put more than one statement on a line. The only exception is the 'for' statement where the initial, final, and loop statements are contained on a single line:

```fortran
for ( i = 1; i <= maxint; i = i + 1 )
```

The 'if' statement is not an exception: the conditionally executed statement goes on a separate line from the conditional expression.
4.2. Horizontal Spacing

(1) A space should be used to separate parentheses from conditional expressions. For example:

```
if ( getlin( line, fd ) != EOF )
  while ( line(i) != EOS )
...
```

When parentheses are used to group arithmetic expressions, no spaces are used for separation. However, spaces are used to surround the operators:

```
(a + b) / ((MAXSIZE * d) / g)
```

(2) Continuation lines should line up with the part of the preceding line they continue (lines must be kept shorter than 80 characters):

```
a = (b + c) *
  (d + e)

call read( a, b, c,
            d, e )
if ( (a == b) &
     (c >= d) )
...
```

(3) Spaces are not used in array references. For example:

```
x = iarray(5)
call gronk( iarray(5 + length), ind )
```

(4) Definitions that need to be continued on another line should precede each continuation line with a single tab character:

```
define(DEF_NAME,
         first line
         second line
)
```

4.3. Indentation

(1) Indentations are made every four characters (i.e., 1, 5, 9, ...).
(2) Outer block starts in column 1. The outer block consists of program and subroutine declarations and documentation.

(3) Indent after each of these sections:
   a) program, subroutine, function declarations, NOIMPLICIT, argument declarations
   b) conditionals (see below)
   c) looping constructs
   d) switch statements
   e) case labels

(4) The 'else' of a conditional has the same indentation as the 'if'. Thus the form of the conditional is:

   if ( <condition> )
   {
   <statements>
   }

   else
   {
   <statements>
   }

(5) Comments have the same indentation as the section of code to which they refer. A space should separate the comment text from the comment character.

(6) Section braces '{' and '}' have the same indentation as the code they enclose.

4.4. Miscellaneous

(1) All variable names, routine names, and keywords are in lower-case only and should be 6 characters or less. Only names of constant values are in uppercase (see below), and are not restricted to 6 characters.

(2) Use the ratfor graphics for operators and relations (<, >, ==, &, |, !, ...).

5. Constant Definitions

Program constants should follow the following conventions:

(1) All constant values are given names with the ratfor 'define' statement.

(2) Names of constant values are in upper-case only. (All other names are in lower-case only.)
(3) Use underlines to make names readable.

(4) Comment the definitions liberally. Line the comments up using tab characters.

(5) Separate out related definitions, such as data structure offsets or values returned from routines, with blank lines and comments.

(6) Values or bits within words in data structures should be defined immediately below the definition of the offset of that word, separated out with a blank line.

(7) Prefix data structure offsets with a mnemonic for that data structure.

(8) Prefix values of a particular class (e.g. possible values of a particular word in a data structure) with a mnemonic for that class.

(9) Avoid abbreviations, except in the data structure and value prefixes defined above. If abbreviations must be used, use them consistently.

The following example illustrates the above conventions:

```
# status values returned by buffer library routines

define(BUFFER_OK, 0)  # operation successful
define(BUFFER_EMPTY, 1) # no data in buffer
define(BUFFER_FULL, 2)  # no free space in buffer

# contents of buffer descriptor

define(BUF_ADDRESS, 1)  # address of data buffer
define(BUF_LENGTH, 2)    # number of words in buffer
define(BUF_STATUS, 3)    # current buffer status

# buffer status values

define(BUF_STS_ERROR, 1)  # buffer invalid
define(BUF_STS_INUSE, 2)  # buffer in use
define(BUF_STS_FREE, 3)   # buffer available

define(BUF_OWNER, 4)     # current owner of buffer
```

SEE ALSO

libconv(info), manual(info)
AUTHOR
    Dave Wilner and Theresa Breckon
NAME
filedirs - Lower-level file directory layout

SYNOPSIS
(see below)

DESCRIPTION

This entry describes the conventions to be used for directory creation and usage. These directories are all defined at the same level -- which is to say that they are all defined directly under a single directory. However, this set of directories may appear at any level of the hierarchy, so long as every directory in that set is defined at the same level. These directories are:

src - A directory containing sources of programs.

bin - A directory containing binary files produced from files in src.
This directory may also contain shell scripts that are executable in the same way as tools.

incl - A directory containing ratfor include files (or analogous files for other languages) for programs in src.

proc - A directory containing procedures (scripts) to compile code contained in src onto bin or lib.

man - A directory of manual entries (usually for programs in src.)

lib - A directory of libraries used by or created by the programs in src.

oldbin - When crucial tools are modified, it is wise to have a backup of the binary file. For this reason, a backup directory, oldbin, is defined at the same level as bin.

oldlib - When crucial libraries are modified, it is wise to have a backup of the library file. For this reason, a backup directory, oldlib, is defined at the same level as lib.

test - A directory of test programs, test cases and scripts used to test the programs in bin or lib.

The conventions for using the preceding layout are as follows:

(1) The source code for programs are kept in src directories. (See Files(0)).

(2) Corresponding binary files are kept in the bin directory which is on the same level as the src directory. When a file is installed in bin, the old version of the file should be saved in oldbin.
(3) Include files are kept in the incl directory which is on the same level as the src directory.

(4) Installation procedures (they are really "sh scripts" but we'll call them "procedures" for historical reasons) are in a directory proc at the same as level as src. They have the same name as the program, except the extension is '.p'. (See Files(0)).

(5) Manual entries for the programs on src are kept in the man directory which is on the same level as the src directory.

(6) Library routines produced or used by programs in src are kept in the lib directory which is on the same level as the src directory. When a file is installed in lib, the old version of the file should be saved in oldlib.

(7) Test programs, test cases and test scripts should be kept in the test directory for programs on bin or libraries in lib.

The source of a program should never mention the full pathname of any file, with the exception of /dev files (devices). Installation procedures should never mention full pathnames.

SEE ALSO
files(info) toplevel(0)

AUTHOR
NAME
files - Files used by the Software Tools on the Vax

SYNOPSIS
(see below)

DESCRIPTION
This entry attempts to collect in one place a variety of lore regarding the Software Tool's use of the Vax file system.

1. File Names

File names can consist of 1 to 8 simple names separated by "/" (slashes). Simple names can consist of 0 to 9 alphanumeric characters, followed by a 0 to 3 alphanumeric extension, followed by an optional version number. (The filename, however, must contain at least one character.) Examples:

```
  a.b .c 12b.xyz;3 (the same as 12b.xyz.3)
```

Case is insignificant in filenames (e.g., "A" is the same file as "a").

There are also some names which are special if they start a filename:

```
  .. " Refers to the current working directory.
  ./ " Refers to the parent directory of the current working directory.
  ~/ " Refers to the home (login) directory.
  */ " Means that the first simple name of the filename is to be used as the volume name on which the rest of the filename resides.
```

Example:

```
  */dra0/usr/src/sh
```

In addition, "../" has a special meaning as a simple name inside a file name, referring to the parent directory of the part of the file name preceeding it. For example, if your login directory is /mnt/joey and your current working directory is /usr/new, then the name

```
  ../src/pr
```

Would refer to /usr/new/src/pr

```
  ../bin/pr
```

Would refer to /usr/bin/pr

```
  ~/src/pr
```

Would refer to /mnt/joey/src/pr

```
  */dra0/mnt/harold
```

Would refer to the file /mnt/harold on the volume DRA0

```
  ~/../jill/pr
```

Would refer to /mnt/jill/pr
2. Reserved Names

The following is a partial list of the names built into various tools.

- `login.com` - DCL command file executed at login time
- `login.rc` - Tools script file executed during resetenv
  (see resetenv(tool) and environ(file))
- `LIB/termcap` - terminal capabilities file used by tset, vi, etc.

3. File Extensions

In order to determine the purpose of a file without looking at its contents, file extensions are used. A file extension is the addition of a '.x' to the normal file name, where 'x' is a letter identifying the use of the file. File extensions are not enforced by the tools (as in UNIX) but are seen only as part of the filename.

Extensions currently standardized by the tools are:

A file extension is generated by:

1. Given a file name (the last simple name of a full path name), remove its extension if there is one.
2. Append the extension.


Each file has permissions which determine what access is allowed by everyone including the owner of the file. (See chmod(tool) and ls(tool) for a discussion of the various permissions). Files are created with "read" permission (but this can be changed using CHMOD). When a tool opens a file for reading, it specifies "Concurrent Read" (which means there may be an arbitrary number of users simultaneously reading the same file). When a file is opened for writing (or creation) "Concurrent Write" is specified (which means that other users can have the file open for read or write.) Thus, strange results will occur if two or more programs attempt to write on the same file at the same time.

SEE ALSO
- paths(info), environ(file), resetenv(tool)
AUTHOR
Van Jacobson & Bob Upshaw
NAME
index - kwic index to Software Tools

SYNOPSIS
The following is a 'Keyword in Context' (KWIC) index of all of the programs in
the RTSG Software Tools programmers manual.

abort - abort a script
 abort - abort a script
finger - list information about people logged on the
fyi - gives information about the system
whois - print out information about users
chmod - change file access protections
index for indexed-sequential access
isam - generate
uniq - strip adjacent repeated lines from a
write - write to another user
 ar - archive file maintainer
 ar - archive file maintainer
asplit - split corrupted archive file substituent files
who - who is on and what are they doing
arguments for a tool
args - use standard input as arguments for a tool
 echo - echo command line arguments
args - use standard input as arguments for a tool
file substituent files
asplit - split corrupted archive
files/directories
backup - backup specified
 upsetape - list contents of backup tape
diff - list differences between two files.
 ld - link program and produce binary
patterns
 fb - search blocks of lines for text
 dc - desk calculator
files
cat - concatenate and print text
itself for awhile
sleep - cause a process to suspend
 only) (vax only)
back - back up directory trees (vax
 only)
o - overstrike - convert backspaces into multiple lines
files/directories
backup - backup specifically

lstape - list contents of backup tape

changes - show changes in the
on-line documentation

ch - make changes in text files

documentation
 changes - show changes in the on-line
tr - character transliteration
dspc - display all characters in a file
postm - check for new mail
lock - check out files or directories

- return files or directories

protelions

cleanup - Clean up files in directory
file structure  cleantree - Cleanup subtree of
directory  cleanup - Clean up files in
structure  cleantree - Cleanup subtree of file

pack - pack words into columns

files

tooldef - DCL command to set up the tools
exit - terminate sh command file
functions)

sh - shell (command interpreter with history
 echo - echo command line arguments
environment untooldef - DCL command to remove the tools
go to sh command transfer
if - conditional sh command

- repeatedly execute a shell command

- a way to repetitively invoke a command

del - execute DCL commands

comm - print lines common to two files

arp - compare two files
fc - fortran compile and link
pc - pascal compile and link
rc - ratfor compile and link

source

yacc - compile and optionally link yacc
rat4 - ratfor compiler
c oppress - compress input files
duse - Compute disk space use
cat - concatenate and print text files
if - conditional sh command

user

confirm - get confirmation from

- copy directory and all files contained

lstable - list contents of backup tape
lstable - list contents of lock file
kwic - make keyword index

lines

os - overstrike - convert backspaces into multiple
spaces

tenab - convert spaces to tabs and
detab - convert tabs to spaces

contained

optree - copy directory and all files
crt - copy files to terminal
op - copy files

output

head - copy first lines of files to
and named files  tee - copy input to standard output
substituent files asplit - split corrupted archive file
we - count lines
          cp - copy files
          cpress - compress input files
files contained optree - copy directory and all
xref - make a cross reference of symbols
crt - copy files to terminal
standard input crypt - crypt and decrypt
          date - print the current date
          pwd - print the user's current working directory
          field - manipulate fields of data
date - print the current date
          - search a file for "matching" dates
finddate
dc - desk calculator
tools environment tooldef - DCL command to set up the
dcl - execute DCL commands
environment unttooldef - DCL command to remove the tools
dcl - execute DCL commands
generate include file for dec symbols
          crypt - crypt and decrypt standard input
            nrddecyn
macro - process macro definitions
          dc - desk calculator
detab - convert tabs to spaces
typeof - determine the type of a file
diff - list differences between
two files.
diff - list
dates
diff - list
dircles checked out
unlock - return files or directories
backup - backup specified files/directories
lock - check out files or directories
- restore specified files/directories
contained
          optree - copy directory and all files
is - list file or directory information
istree - list file system directory sub-tree
          back - back up directory trees (vax only)
          cd - change working directory
          cleanup - Clean up files in directory
          mkdir - make directory
print the user's current working directory
          pwd -
          rm - remove directory
          pq - show disk quota and space used
duse - Compute disk space use
dspc - display all characters in a file
ruler - display ruler on terminal screen
the tools specified man - display the manual entries for
usage - display usage of a tool
f - format an RISG standard
document
- show changes in the on-line
documentation
changes
help - more on-line
documentation
- who is on and what are they
doing who
a file
dsppc - display all characters in
duse - Compute disk space use
arguments
echo - echo command line

ed - execute visual editor
ed - text editor
esedit - stream editor
ed - execute visual editor
ed
and spaces
entab - convert spaces to tabs
man - display the manual
entries for the tools specified
m - format a programmer's manual
entry
command to set up the tools
environment tooldef - DCL
DCL command to remove the tools
environment untooldef -
spell - find spelling
errors
repeat - repeatedly execute a shell command
dcl - execute DCL commands
edt - execute visual editor
exec - execution of a process
exit - terminate sh command
file
logout - exit shell
expand - uncompress input files
incl - expand included files
document
fb - search blocks of lines for
fc - fortran compile and link
text patterns
f - format an RISG standard
data
field - manipulate fields of data
chmod - change file access protections
findlfile - search a file for "matching" dates
mkdecsym - generate include file for dec symbols
find - search a file for text patterns
split - split a file into pieces
ar - archive file maintainer
ls - list file or directory information
cleantree - Cleanup subtree of
file structure
asplit - split corrupted archive file substitute files
lstree - list file system directory sub-tree
mww - write file to tape
lpr - queue a file to the printer
path
lookfor - look for a file using a standard search
- display all characters in a file
dspc
exit - terminate sh command
file
lslock - list contents of lock
file
print specified lines/pages in a file
pl -
strings - find the strings in a file
tail - print last lines of a file
typeof - determine the type of a file
adjacent repeated lines from a file
uniq - strip
reverses the order of lines in a file.
revl -
cptree - copy directory and all files contained
archive file substitute files
pr - paginate files for printing
ntr - read files from tape
cleanup - Clean up files in directory
unlock - return files or directories checked out
lock - check out files or directories
head - copy first lines of files to output
crt - copy files to terminal
archive file substitute files
asplit - split corrupted
files
concatenate and print text files
cat -
changes in text files
ch -
compare two files
amp -
print lines common to two files
comm -
copy files
files
compress input files
cp -
compress input files
expand - uncompress input files
expand - uncompress input files
includ - expand included files
incl -
laminate files
lam -
make files
mk -
move files
mv -
remove files
rm -
sort - sort and/or merge text files
diff
to standard output and named files
tree - copy input
tree -
list differences between two files.
backup - backup specified files/directories
restore - restore specified files/directories
find - search a file for text
find -
spelling errors
spell -
find the strings in a file
strings -
finddate - search a file for
about people logged on the
finger - list information
head - copy first lines of files to output
by prompting user for
head - copy first lines of files to output
form - produces form letter
user for form - produces form letter by prompting
entry
m - format a programmer's manual
f - format an RISC standard document
mmp - format mmp
tbl - tool to format tables
index(info)

February 15, 1985

index(info)

roff - format text
makel - multicolum formatting
fc - fortran compile and link
interpreter with history functions) sh - shell (command
the system fyi - gives information about
symbols mkdecsym - generate include file for dec
indexed-sequential access isam - generate index for
confirm - get confirmation from user
system fyi - gives information about the
goto - sh command transfer

Ingen - translates LR grammar into parse tables.
to output head - copy first lines of files
documentation help - more on-line
shell (command interpreter with
history functions) sh -
if - conditional sh command
incl - expand included files

mkdecsym - generate include file for dec symbols
incl - expand included files
access isam - generate index for indexed-sequential
kwic - make keyword in context index
isam - generate index for indexed-sequential access
logged on the finger - list information about people
fyi - gives information about the system
whois - print out information about users
ls - list file or directory information
ps - list process status information
args - use standard input as arguments for a tool
compress - compress input files
expand - uncompress input files
nared files tee - copy input to standard output and
-crypt and decrypt standard input crypt
prompt - Prompt a User for Input
functions) sh - shell (command interpreter with history
sched - a way to repetitively invoke a command
indexed-sequential access isam - generate index for
- cause a process to suspend itself for awhile
kwic - make keyword in context index
sleep
kill - kill a subprocess
kill - kill a subprocess
index kwic - make keyword in context
-unrotate lines rotated by kwic
unrot
lam - laminate files
 lamin - laminate files
tail - print last lines of a file
binary
ll - print line lengths
form - produces form letter by prompting user for
libup - tool to update libraries
   - list the program names in a library
   slib - tool to update libraries

- echo - echo command
- changes - show changes in the
- help - more on-
- 11 - print line lengths
- suggest - make an on-
- comm - print lines common to two files
- fb - search blocks of lines for text patterns
- uniq - strip adjacent repeated lines from a file
- revl - reverses the order of lines in a file.
- tail - print last lines of a file
- head - copy first lines of files to output
- unrot - unrotate lines rotated by kwic

- convert backspaces into multiple lines
- rev - reverse lines
- wc - count lines
- pl - print specified lines/pages in a file
- ld - link program and produce binary
- yacc - compile and optionally link yacc source
- fc - fortran compile and link
- pc - pascal compile and link
- rc - ratfor compile and link

- lip - list program

- logged on the
- finger - list information about people
- 1stape - list contents of backup tape
- lslock - list contents of lock file

- files.
- diff - list differences between two
- 1s - list file or directory
- sub-tree
- 1stree - list file system directory
- ps - list process status information
- lip - list program

- library
- slib - list the program names in a
- 11 - print line lengths

- directories
- lslock - list contents of
- list information about people
- logged on the
- logout - exit shell

- search path
- lookfor - look for a file using a standard
- a standard search path
- printer
- lpr - queue a file to the

- lrgen - translates LR grammar into parse tables.
- - translates yacc source to
- information
- in a library
- slib - list the program names
index(info)  February 15, 1985  index(info)

file
lslock - list contents of lock

tape
lstape - list contents of backup

directory sub-tree
lstree - list file system

tree entry
m - format a programmer's manual

definitions
macro - process macro

        macro - process macro definitions
        roffmac - roff macro processing for roff
        mail - send mail to other users

postmac - check for new mail
rdmail - read your mail
ar - archive file maintainer

symbols
xref - make a cross reference of
suggest - make an on-line suggestion
ch - make changes in text files
mkdir - make directory
mk - make files
kwic - make keyword in context index

for the tools specified
man - display the manual entries
for the tools specified
man - display the manual entries for the tools
finddate - search a file for matching dates

        finddate - search a file for matching dates
        mcol - multicolumn formatting
        mmop - format mmop

        mmop - format mmop
        sort - sort and/or merge text files
        mk - make files

for dec symbols
mkdirsyn - generate include file
mkdir - make directory

        mrdt - read a modcomp tape
mrdt - write a modcomp tape
        mrdtr - read modcomp tree
mv - move files

        mv - move files
        mtr - read files from tape
        mtw - write file to tape

        mcol - multicolumn formatting
        - convert backspaces into multiple lines
        os - overstrike

input to standard output and
unnamed files

        lnslib - list the program names in a library
        postma - check for new mail
        changes - show changes in the on-line documentation
        help - more on-line documentation
        suggest - make an on-line suggestion
        - back up directory trees (vax only)
        yacc - compile and optionally link yacc source
revl - reverses the order of lines in a file.
backspaces into multiple lines os - overstrike - convert output and named files
tee - copy input to standard tee - copy first lines of files to output
- copy first lines of files to output
head
backspaces into multiple lines os - overstrike - convert backspaces pack - pack words into columns
pack - pack words into columns
pl - print specified lines/pages in a file
pr - paginate files for printing
- translates LR grammar into parse tables lrgen
pc - pascal compile and link
a file using a standard search path lookfor - look for
search blocks of lines for text patterns fb - find - search a file for text patterns
- list information about people logged on the finger
split - split a file into pieces
in a file
pl - print specified lines/pages
pr - paginate files for printing
pq - show disk quota and space
- paginate files for printing
pr - paginate files for printing
tail - print last lines of a file
ll - print line lengths
comm - print lines common to two files
users
whois - print out information about
file
pl - print specified lines/pages in a
print text files
cat - concatenate and
- print the current date
date
print the user's current working
lpr - queue a file to the printer
pr - paginate files for printing
macro - process macro definitions
ps - list process status information
awhile
sleep - cause a process to suspend itself for
resume - resume a suspended process
suspend - suspend a running process
timer - time execution of a process
roff - process for rtf
ld - link program and
produce binary prompting user for form - produces form letter by
ld - link program and produce binary
lslib - list the program names in a library
lip - list program
n - format a programmer's manual entry
prompt - Prompt a User for Input
prompt - Prompt a User for Input
- produces form letter by prompting user for form
drmod - change file access protections
information ps - list process status
working directory pwd - print the user's current
lpr - queue a file to the printer
pq - show disk quota and space used
rat4 - ratfor compiler
rc - ratfor compile and link
rat4 - ratfor compiler
rc - ratfor compile and link
rdmail - read your mail
rm - read a modcomp tape
rtr - read files from tape
rmdir - read modcomp tree
rdmail - read your mail
xref - make a cross reference of symbols
rmdef - remove directory
rm - remove files
untooldef - DCL command to remove the tools environment
shell command repeat - repeatedly execute a
uniq - strip adjacent repeated lines from a file
command repeat - repeatedly execute a shell
sched - a way to repetitively invoke a command
files/directories restore - restore specified
files/directories restore - restore specified
process resure - resume a suspended
checked out unlock - return files or directories
rev - reverse lines
rev - reverse lines
file. rev1 - reverses the order of lines in a
lines in a file. rev1 - reverses the order of
rm - remove files
rmdef - remove directory
rmdef - remove directory
rm - read a modcomp tape
rm - read modcomp tree
rm - read modcomp tree
rm - format text
roff - macro processing for roff
unrot - unrotate lines rotated by kwic
f - format an RISC standard document
terminal screen ruler - display ruler on
ruler - display ruler on terminal screen
suspend - suspend a running process
invoke a command sched - a way to repetitively
- display ruler on terminal screen
aborted - abort a script
finddate - search a file for 'matching'
find - search a file for text patterns
patterns
fb - search blocks of lines for text
look for a file using a standard
- generate index for indexed
- set up the tools environment
mail - sequential access
- tooldef - DCL command to
with history functions)
exit - terminate
sh - shell (command interpreter
if - conditional
history functions)
repeat - repeatedly execute a
logout - exit
documentation changes -
sh - shell (command interpreter with
suspend itself for awhile
files
sort - sort and/or merge text
sort - sort and/or merge text files
tsort - topologically
temporary
yacc - translates yacc
compiler or optionally link yacc
compile and optionally yacc source to lrgen source
source
- duser - Compute disk space use
pq - show disk quota and space used
tenab - convert spaces to tabs and spaces
detab - convert tabs to spaces
- convert spaces to tabs and spaces
backup - backup specified files/directories
restore - restore specified files/directories
pl - print specified lines/pages in a file
the manual entries for the tools specified
man - display spell - find spelling errors
spell - find spelling errors
split - split a file into pieces
split - split a file into pieces
substitute files asplit - split corrupted archive file
f - format an RTSG standard document
a tool args - use standard input as arguments for
crypt - crypt and decrypt standard input
tee - copy input to standard output and named files
- look for a file using a standard search path lookfor
ps - list process status information
edit - stream editor
file
strings - find the strings in a
strings in a file
from a file uniq - strip adjacent repeated lines
- Cleanup subtree of file structure cleantree
- list file system directory subtree lstree
  kill - kill a subprocess
- split corrupted archive file
  substituent files asplit
  cleantree - Cleanup subtree of file structure

  suggestion
  suggest - make an on-line suggestion
  suspend - suspend a running process
  suspend a process to suspended process
  process
  suspend - suspend a running

  - generate include file for dec symbols
  tsort - topologically sort symbols
  xref - make a cross reference of lstree - list file symbols
  lstree - list file system directory sub-tree
  - gives information about the system
  tbl - tool to format tables
  translates LR grammar into parse tables.
  - tools translate tables.
  ingen -
  - tools translate tables.
  entab - convert spaces to tabs and spaces
  detab - convert tabs to spaces
  file
  tail - print last lines of a tape
  - list contents of backup
  ntrr - read files from tape
  mftw - write file to tape
  wroxt - read a modcrop tape
  wrxdt - write a modcrop tape
  - tools translate tables.
  tbl - tool to format tables
  output and named files
  tee - copy input to standard
  - rules translate tables.
  ruler - display ruler on terminal screen
  - tools translate tables.
  crt - copy files to terminal
  - tools translate tables.
  exit - terminate sh command file
  exit - terminate sh command file
  ed - text editor
  - tools translate tables.
  cat - concatenate and print text files
  - tools translate tables.
  ch - make changes in text files
  sort - sort and/or merge text files
  - tools translate tables.
  fb - search blocks of lines for text patterns
  find - search a file for text patterns
  - tools translate tables.
  roff - format text
  who - who is on and what are they doing
  - tools translate tables.
  timer - time execution of a process
  process
  timer - time execution of a
  tbl - tool to format tables
  - tools translate tables.
  libup - tool to update libraries
  input as arguments for a tool
  usage - display usage of a tool
  - tools translate tables.
  usage - display usage of a tool
  up the tools environment
tooldef - DCL command to set
  - DCL command to set up the tools environment

- tooldef
- DCL command to remove the tools environment untooledf
  - tools environment
  - untooledf
  - manual entries for the tools specified
  - man - display
tsort -
  - topologically sort symbols
tr - character transliteration
goto - sh command transfer
tables.
irlgen -
  - translates LR grammar into parse source
yacc - translates yacc source to irlgen
tr - character transliteration
  - list file system directory sub-
tree
  - tree
back - back up directory
  - trees (vax only)
symbols
  - to set
  - symbol
  - type of a file
expand -
  - type of 
  - determine the type of a file
lines from a file
  - uncompress input files
directories checked out
  - uniq - strip adjacent repeated
by kvic
unlock - return files or
unrot - unrotate lines rotated
unrot - unrotate lines rotated by kvic
remove the tools environment untooledf - DCL command to
  - back - back up directory trees (vax only)
cleanup - Clean up files in directory
tooldef - DCL command to set up the tools environment
libup - tool to update libraries
usage - display usage of a tool
  - display usage of a tool
duse - Compute disk space
  - use
pq - show disk quota and space
  - used
form letter by prompting user for form - produces
prompt - Prompt a User for Input
confirm - get confirmation from user
  - write - write to another user
  - pxr - print the user's current working directory
mail - send mail to other users
  - print out information about users
  - whois
  - lookfor - look for a file using a standard search path
  - back - back up directory trees (vax only)
edt - execute visual editor edt
command
  - sched - a way to repetitively invoke a
  - wc - count lines
who - who is on and what are they doing
  - who - who is on and what are they
  - whois - print out information
  - who - what are they doing
pack - pack words into columns
  - words into columns
summary - change working directory
pwd - print the user's current working directory
write - write to another user

wmdt - write a modcmp tape
mtw - write file to tape
write - write to another user

symbols
link yacc source
yaclr - translates yacc source to lrgen source
- compile and optionally link yacc source
  yacc
  - compile and optionally link yacc source to lrgen source
yaclr - translates yacc source
NAME
libconv - Library Conventions

SYNOPSIS
src/xxxlb - library source code incl/xxxdef - include file for library
definitions incl/xxxcom - include file for common block definition
lib/xxxlb - compiled library object proc/xxxlb.p - installation procedure
man/lib/xxxlb.m - source of manual entry for library
man/lib/xxxlb - manual entry for library

DESCRIPTION
A library is a collection of routines which are very closely related in function. Usually they form a package of operations on some common object. Examples are a symbol table manipulation library, a dynamic memory management library, or a database access library.

Each such library is identified by a unique two or three character mnemonic. Examples are 'tbl' for the symbol table manipulation library, 'ds' for the dynamic memory management library, and 'db' for the database access library. The name of the library as a whole (and of the object library in the lib directory) consists of the library mnemonic followed by the two character mnemonic 'lb'. The names of the previous examples would be 'tbllb' 'dslb' and 'dblb'.

If there are several files that make up a single library, another character may be used to identify the file. In this scheme, the two database library files would be called 'db1lb' and 'db2lb'.

If possible, the names of all the routines in a library should begin with the library mnemonic. For example, 'dsaput', 'dsfree', and 'dsget' would be routines in the dynamic memory allocation library. 'dbget', 'dbput', and 'dbproj' would be the names of routines in the database library. These names should also be restricted to 6 characters.

Each library has the following set of standard files associated with it. For this example, the library mnemonic is 'xxx':

src/xxxlb - contains the source code of the entire library.
incl/xxxdef - contains the constant definitions that are used by the library routines. If the contents of an include file are internal to the library, i.e., are not needed by any other program or library, an include file should not be used. The contents of the include file should just be incorporated into the source code. However, if the definitions are shared by other source files or are needed by users of the library, they should be placed in the include file 'incl/xxxdef'.

- 1 -
incl/xxxcom - contains the ratfor declarations for any common blocks used by the library. The common block names should begin with the library mnemonic. If the common blocks are internal to the library, i.e., are needed by this one library only, commons can be declared using a definition instead of an include file. For example:

```ratfor
define(DS_COMMON,
       integer dsnmem(DS_MEMSIZE)
       common /dscom/ dsnmem
    )
```

The definition "DS_COMMON" would be used in place of the "include xxxcom". Notice the layout of the above definition. This layout is part of the coding conventions and is described in more depth in `codeconv(info)`.

lib/xxxlb - contains the compiled object library of src/xxxlb.

proc/xxxlb.p - contains the installation procedure for the library. It compiles the source and generates the object library onto lib/xxxlb.

man/lib/xxxlb.m - contains the source of the manual entry for the library, and man/lib/xxxlb contains the resulting manual entry.

FILES

SEE ALSO
files(info), filedirs(info), codeconv(info), manual(info)

AUTHOR
David Wilner and Theresa Breckon
NAME
manual - Programmer's Manual

SYNOPSIS
(see below)

DESCRIPTION

This document describes how to maintain and add entries to the online
programmer's manual. Whenever the system is changed, the associated
manual entry(s) will be updated to reflect the change, in order to make the
information immediately available to all users.

A shell script for producing manual entries is available [see m(tool)] together
with a set of manual oriented macros [see manmac(lib)]. A prototype manual
entry (suitable for 'fill-in' construction of a new entry) is available in
/usr/lib/manproto.

1. Manual Structure

The manual is divided into several functionally oriented chapters. The
chapter is identified by a mnemonic which appears in parenthesis next to the
name of each entry in the chapter:

  info - General Information
  tool - Software Tools
  lib - Library Routines
  file - Special Files and Devices
  data - Database Entries and Data Structures
  maint - Maintenance and Operation
  analysis - Data Analysis and Display
  tutorial - Tutorials or extended manuals for
          programs described in the tools section

Each chapter contains alphabetically ordered entries describing programs,
functions, etc. Each entry is composed of several sections (indicated by
headings in capital letters). These sections have specific purposes and for­
mats and are labeled as follows:

  NAME - name and short description of object
  SYNOPSIS - summary of object usage
  DESCRIPTION - full description of object
  FILES - files used by object
  SEE ALSO - list of related manuals
  IMPLEMENTATION - hints for implementors
  DIAGNOSTICS - diagnostic messages given by object
  AUTHOR - inventor of object
  BUGS - list of object's bugs

The above order should be adhered to in the manual entry. These sections
are described more fully below.
2. NAME

This section has the name of the object being described in the entry and a one line description of the object. For example:

```plaintext
cmp - compare two files
```

3. SYNOPSIS

This section gives a summary of how to use a command, system call or library routine. Optional arguments or flags are enclosed in square brackets ([..]). Repeated arguments are indicated by an elipsis (...). For example:

```plaintext
ls [-l] [file ...]
```

Descriptions of flags should follow the summary of a command. Declarations of parameters and function types should follow the summary of system calls and/or library routines. The synopsis for a tool should look like:

```plaintext
more [-<n>] [-cls] [file] ...
  -<n>: set window size to <n>
  -c: clear screen when drawing new screenfuls
  -l: display with line numbers
  -s: compress adjacent blank lines (squeeze)
```

Below is an example of a synopsis for a manual in the `lib` section describing a library of routines, `src/extlb` (this library is fictional). The synopsis for system calls and library routines should contain an example of the call followed by the declarations for that call. If a function returns a few specific values, like YES and NO, EOF and len, etc., these values should be specified in the example of the function call:

```plaintext
call extadd( fname, ext )
YES/NO = extext( fname, ext )
ind = extget( fname )
```

```plaintext
integer ind, extget, extext
character fname(), ext()
```

4. DESCRIPTION

This section is the main body of the entry and describes the object's use, purpose, idiosyncracies and any thing else the author feels necessary. Describing how a program or routine works is seldom useful information in a Programmer's Manual and should not be done in this section. (See IMPLEMENTATION section.)
5. FILES

This section lists any files or system data structures used by the object being described. The description should contain the name of the file and a one line description. For example:

/misc/lib/termcap - terminal capabilities data base

6. SEE ALSO

This section points the reader to other manual entries or external documentation. Manual entries should be in order of relevance. Other documents should follow the manuals and be in bibliography form. For example:

m(tool), roff(tool), rofmac(lib), manmac(lib)
st.2.2.5 - "Standard Document Preparation"

7. IMPLEMENTATION

This section should describe briefly any implementation notes which may be useful to an implementor. The contents of this section are suppressed in the Programmer's Manual since this information is only useful to implementors.

8. DIAGNOSTICS

Explains any diagnostic or error messages. NOTE: messages which are self-explanatory are not listed. However, messages which may be confusing to the casual user should be described.

9. AUTHOR

In addition to giving credit (or blame) this section is meant to suggest at least one person that users can contact for additional information or in the event of problems.

10. BUGS

This documents any known bugs and shortcomings of the object described, including features which should be added.

11. Manual Storage

Manual entries reside in directories named '/Project/man/Section'. 'Project' is the standard directory for the associated project. For example, 'usr'
for all the software tools and 'db' for the Data Base code. 'Section' is the
chapter of the manual entry. Each file in the directory is a manual entry.

Each manual exists on two different files inside the directory. The roff output
exists on a file with the same name as the manual. The roff input file name is
the same as the roff output file name with an extension of "m". For example,
the roff input of this manual would be in the file "/usr/man/info/manual.m"
and the roff output would be in "/usr/man/info/manual".

12. Making Changes

All manual changes are made to the "m" manual files in your own file space,
never to the actual manual entry. When changing an existing entry, one
should make a copy of the "m" manual file to use as a starting point (when
creating a new entry, a prototype on /usr/lib/manproto can be copied for
use as a starting point).

FILES
/usr/lib/manproto, /usr/lib/manmac, /usr/lib/rofmac

SEE ALSO
m(tool), roff(tool), manmac(lib), rofmac(lib)

AUTHOR
Van Jacobson and Theresa Breckon
NAME
paths - Software Tools' Search Paths

SYNOPSIS
(See Below)

DESCRIPTION
This document describes 'search paths' under Software Tools. It also mentions the search path assumptions built into various tools (the shell, ratfor, ld, etc.).

A search path is simply a list of places to look for some file. The places usually depend on the tool being used (e.g., sh) and your preferences. For example, if you are doing program development for 'nbs', you might want the shell to search

1. the current working directory (to pick up things that you're working on)
2. your 'bin' directory (to pick up any of your 'customized' commands)
3. the '/nbs/bin' directory (to pick up the project's standard commands & program installation procedures)
4. and, finally, the '/usr/bin' directory to pick up standard program development commands.

However, you would probably want the ratfor compiler to search for "include" files in

1. your 'incl' directory
2. the '/nbs/incl' directory
3. the '/usr/incl' directory.

Each tool that uses a search path uses an environment variable to determine that path [see environ(file)]. For example, the shell gets its path from the environment variable PATH. The shell search path described above would be setup by the following setenv call.

    setenv PATH ". ~/bin /nbs/bin /usr/bin"

Each command given the shell which isn't a complete filename (a complete filename starts with "/", "./", ".../", or "~/") would be searched for in each directory named in PATH. For example, if you typed "gronk" and PATH was set to the value shown above, the shell would execute the first found of:

- "gronk" in the current working directory
- "gronk" in your "bin" directory
- "/nbs/bin/gronk"
paths (info) March 29, 1985 paths (info)

"/usr/bin/gronk"

Some of the most important search path environment variables are:

- PATH - used by the shell to look up commands.
- INCL - used by ratp1 to find 'include' files.
- LIB - used by the linker, ld, to find binary library files.

SEE ALSO

- environ(file), setenv(tool), unsetenv(tool), resetenv(tool), files(info), ld(tool), ratp1(tool), sh(tool), splib(tool)

AUTHOR

Van Jacobson & Bob Upshaw with environments added by Theresa Breckon.
NAME
prims - list of available primitive routines

SYNOPSIS
(See below)

DESCRIPTION
Below is a list of all routines and libraries available to the user.

1. Portable Primitives

The following is a list of portable primitives. These are the primitives that should be used by people writing portable tools.

acopy - copy n characters from file1 to file2
addset - put character into array if it fits
alldig - return YES if string is all digits
amove - move (rename) file1 to file2
appchr - append character onto the end of a string
appstr - append one string to another
bubble - bubble sort an integer array (increasing order)
cant - print 'name: can't open' and aborts execution
c~copy - append character to string at specified
chdir - change working directory
chkdir - check to see if passed file is a directory
close - close (detach) a file
clower - fold passed a character to lower case
cmatch - try to find character in string
concat - concatenate the two passed strings
create - create a new file or overwrite if existing
cstoi - convert string at specified location to integer
cupper - convert character to upper case
cyplb - cypher library (encryption algorithms)
delarg - delete command line argument number 'n'
dospwn - spawns a shell command
der4 - close all open files and terminate program
equal - compare two strings
error - outputs one-line error message and aborts
esc - map element of array into escaped character
evalb - expression evaluation routine library
fcopy - copy one file to another
fnlb - library routines to deal with file names
fntopn - convert file name to path name
fold - convert string to lower case
fsize - determine size of a file (in characters)
fskip - skip n characters on a file
gdate - get current date
getarg - get command line arguments
getc - get character from STDIN
getch - read character from file
getlin - get next line from file
gettyp - get type of file
getwrd - get non-blank work from character array
glname - get user's login name
gowner - get user's owner name
gtime - get current time
guname - get user's real name
index - kwic index to modcorrtp tools
inptr4 - initialize all standard files and common
itob - convert integer to string of base b
itoc - convert integer to character string
itoczf - converts an integer to a character string
length - compute length of string
mailb - mail utility routines
manmac - Macros for Programmer's Manual Entries
markl - determine file position of next record
memlb - ratfor dynamic memory allocation routines
mtlb - mag-tape library
nmfile - get name of file associated with file
open - open an existing file for reading
panic - output a message and stop
prompt - Prompt a User for Input
prtln - print lines to STDOUT with pauses
putc - put character onto STDOUT
putc - put character onto STDAGUT
putch - write character to file
putdec - write integer in specified field width
putint - output integer in specified field
putlin - output a line onto a given file
putstr - output character string in specified field
quelb - integer queue and stack library
query - write a prompt and accept the user response
remark - print single-line message
remove - remove a file
rewind - rewinds file
rndlb - random number library
scopy - copy (sub)string to another string
scratf - get scratch file name based on 'seed'
seek - move read/write pointer on random-access file
shell - shell sort an integer array in increasing order
skipbl - skip blanks and tabs at specified position in a
spawn - execute subprocess
splb - library to use search paths
stcopy - copy (sub)string to another string incrementing
strcmp - compare two strings
tblb - symbol table routines
termin - pick up identifier of read channel to user's
trmout - pick up identifier of write channel to users
trmsiz - get terminal size in lines
tty - determine if file is a teletype/CRT device
type - determine type of character
typlb - determine the basic type of a file
upper - convert string to upper case
wclb - routines to match wildcard characters in
weofbc - write EOF before closing READWRITE file

2. Non-Portable Primitives

These primitives are available on this machine only. They should not be used by people who wish to write portable tools.

cctor - Convert character string to real number
dqlb - (qlb) integer double-ended queue library
fprintf - formatted output to a file
nsklb - library to stack strings
printf - formatted output to STDOUT
putspaces - write a specified number of blanks to STDOUT
qlb - integer queue library
rtoc - Convert a real number to a character string.
rtsgmac - macros to format text
setabort - set abort status
siowner - set internal owner name
sleep - cause a process to suspend itself for awhile
stklb - (qlb) integer stack library
yyplb - yacc parser library

For more information on a specific primitive, say 'man prim', where 'prim' is replaced with the name of the primitive.

SEE ALSO
man(lib)

AUTHOR
Bob Upshaw

BUGS/DEFICIENCIES
Probably a few. Don't hesitate to fill out an SPR.
NAME

tools - List of RTSG Software Tools

SYNOPSIS

(See below) NOTE: tools.p creates the manual entry /usr/man/info/tools.m
The manual entry should not be updated directly.

DESCRIPTION

Below is a list of all tools available to the user using the set of RTSG Software Tools on this machine.

General Tools

abort - abort a script
ac - assemble and link
admin - Create a TCS file
ar - archive file maintainer
args - use standard input as arguments for a tool
asplit - split corrupted archive file substituent files
back - back up directory trees (vax only)
backup - backup specified files/directories
cat - concatenate and print text files
cd - change working directory
ch - make changes in text files
changes - show changes in the on-line documentation
chmod - change file access protections
cleantree - Cleanup subtree of file structure
cleanup - Clean up files in directory
cmp - compare two files
comm - print lines common to two files
confirm - get confirmation from user
cp - copy files
cpress - compress input files
cptree - copy directory and all files contained
crt - copy files to terminal
crypt - crypt and decrypt standard input
date - print the current date
dc - desk calculator
dcl - execute DCL commands
delta - make a TCS delta
detab - convert tabs to spaces
diff - list differences between two files.
dspc - display all characters in a file
echo - echo command line arguments
ed - text editor
edt - execute visual editor edt
entab - convert spaces to tabs and spaces
exit - terminate sh command file
expand - uncompress input files
f - format an RTSG standard document
fb - search blocks of lines for text patterns
fc - fortran compile and link
field - manipulate fields of data
find - search a file for text patterns
finddate - search a file for "matching" dates
finger - list information about current users
form - produces form letter by prompting user
fyi - provides system information from keywords
get - get generation from TCS file
goto - sh command transfer
head - copy first lines of files to output
help - more on-line documentation
if - conditional sh command
includ - expand included files
isam - generate index for indexed-sequential access
kill - kill a subprocess
kwic - make keyword in context index
lam - laminate files
ld - link and produce binary
lex - lexical analyzer generator
libind - create an index of library routines
libup - tool to update libraries
lip - list program
li - print line lengths
lock - check out files or directories
logout - exit shell
lookfor - look for a file using a standard search path
lpr - queue a file to the printer
lrgen - translates LR grammar into parse tables.
ls - list file or directory information
lslib - list the program names in a library
lslock - list contents of lock file
lstape - list contents of backup tape
lstree - list file system directory sub-tree
m - format a programmer's manual entry
macro - process macro definitions
Mail - utility for sending mail to local users
man - display the manual entries for the tools specified
mcol - multicolumn formatting
memo - format memo
mk - make files
mkdecsym - generate include file for dec symbols
mkdir - make directory
Mretry - re-submit mail saved by composition programs
Msg - utility for sending and receiving mail.
Msplt - utility for salvaging message files
mtr - read files from tape
mtw - write file to tape
mv - move or rename files
note - leave a note for a reminder
notify - notify the user of any outstanding notes
os - overstrike - convert backspaces into multiple lines
pack - pack words into columns
pc - pascal compile and link
pl - print specified lines/pages in a file
Postmn - report the presence of mail
pq - print disk quota
pr - paginate files for printing
printenv - list the user's environment
prlabl - format labels for printing
prompt - prompt a user for input
ps - list process status information
pwd - print the user's current working directory
rat4 - translate ratfor into fortran
ratags - create a tags file for a ratfor program
ratfix - fix a source file for the new ratfor
ratp1 - pass one of ratfor to fortran translator
ratp2 - pass two of ratfor preprocessor
rc - ratfor compile and link
repeat - repeatedly execute a shell command
resetenv - re-set your environment
Resolve - resolve mail system user names
restore - restore specified files/directories
resume - resume a suspended process
rev - reverse lines
revl - reverses the order of lines in a file.
rm - remove files
rmdir - remove directory
rmodt - read a modcomp tape
rmodtr - read modcomp tree
roft - format text
roffmac - macro processing for roff
ruler - display ruler on terminal screen
sched - repetitively invoke a shell command
sed - stream editor
setenv - set the value for an environment variable
sh - shell (command interpreter with history functions)
sleep - cause a process to suspend itself for awhile
sort - sort and/or merge text files
spell - find spelling errors
split - split a file into pieces
Stmail - utility for sending mail to other users
strings - find the strings in a file
stty - set terminal characteristics
submit - submit shell commands to a batch queue.
suggest - make an on-line suggestion
suspend - suspend a running process
tail - print last lines of a file
tbl - tool to format tables
tee - copy input to standard output and named files
tooldef - DCL command to set up the tools environment
tr - character transliteration
tsort - topologically sort symbols
typeof - determine the type of a file
uniq - strip adjacent repeated lines from a file
unlock - return files or directories checked out
unrot - unrotate lines rotated by kwic
unsetenv - un-set the value for an environment variable
untooldef - DCL command to remove the tools environment
usage - display usage of a tool
Users - list valid mail users
vi - use vi from the shell
Vmstost - converts the VMS mail to MSG message format
wc - count lines
who - show who's on and what they are doing
whois - print out information about users
wmodt - write a modcomp tape
write - write to another user
xref - make a cross reference of symbols
yacc - translate Yacc program into Ratfor
yacrl - translates yacc source to lrgen source

Data Analysis Tools
1stdif - compute first differences on a file of data
fft - Fast Fourier Transform
genlin - generate points on a polynomial line
genord - generate ordinal values for lines
graph - graph data
histds - graph a histogram
linreg - do a least-squares fit to multivariate data
mkhist - histogram data
npavg - smooth data using a running average
pfit - do a polynomial fit to data
regres - correlate variables by linear regression
stat - compute statistics on a file of data
FILES
   /usr/proc/tools.p - tool to create this manual entry.

SEE ALSO
   man(tools)

DIAGNOSTICS

AUTHOR
   Nancy Travis
NAME
toplevel - Top-level file directory layout

SYNOPSIS
(see below)

DESCRIPTION

This entry describes the top-level directory layout on the RTSG tools systems. There are several sets of directories; the name and usage of each directory in the set is described.

1. User Private File Space

The first set consists of just one directory, - /mnt. This set is put aside for the use of individuals.

/mnt - This directory contains the private space for all users. Underneath this top level, each user's space is allocated in directories with the name /mnt/<username>. Each user has an owner name as defined in the password file. Each user may do as she pleases with this directory. However, it is strongly recommended that all users follow the conventions described in filedirs(0). This is particularly true when modifying and installing programs maintained by a group.

2. Tools Files

The second set of directories is for use by the tools environment. This set contains everything needed to support the tools. It contains no private space, nor contain anything that supports the underlying operating systems.

/usr - This directory contains the tools, tools libraries, compilers and their libraries, user tools such as backup procedures, and so on. All files being added to this directory must go through the appropriate system change procedures.

/etc - This directory contains miscellaneous files (e.g. data bases on users, devices, etc.) This, too, must only be updated by the appropriate system change procedure.

/tmp - This directory contains temporary files (e.g. editor work files, spawn files etc.)

3. Project Files

The next set of directories is designed to support different groups of users who are using the tools to develop products for projects. This includes the
NBS project, and the BART project. Not all top-level directories need exist on every system, but the structure of each instance of such a directory should be the same.

The directories are:

/bart
/nbs

These directories contain those pieces needed to make up the each of these two control systems. It does not contain common system pieces. Each directory is maintained by the project. The underlying structure of these directories is described elsewhere. Objects should only be installed by following the system change procedures defined by the maintainers.

4. Common System Pieces (Modcomp Only)

The following directories are each organized as described in `filedirs(0)`. Each directory contains a set of files for the subsystem given by the name of the directory. The directories, and their contents, are described as they are intended to be. However, since all files have not been transferred to the Tools system, there are some renegades. The renegades come from the old partitions S3L, S4L, and S7L. Their compilation procedures are still in Job Control. These files live in the temporary directory /sys; their organization is a simple copy from the partitions. For example, an object that was in S3L is now in /sys/s3l.

These directories are:

/db - All database pieces reside in the /db directory. This includes all database systems from SEED to the various Run Time Databases.

/network - This directory contains all network pieces, for both Max III and Max IV.

/graphics - This directory contains all pieces needed to make up the graphics systems. This includes the Grinnel Symbiont, and the TCS/AG2 libraries.

/max3 - This directory contains all the pieces needed to make up the Max III system. Objects that are functionally separate from Max III, such as the network, reside in their own directories.

/max4 - This directory contains the pieces need to make up a Max IV system. This includes the system itself, and utilities designed for Max IV only.

/max7 - This directory contains utilities that are common to both Max III and Max IV.
5. Others

/misc - This directory contains public files. There are no controls on this directory, nor any conventions about what should be placed in this directory. The structure of the directory should follow the conventions laid out in filedirs(0). Its function is to hold common pieces that are not a part of the system, nor are really private objects. Examples are the chess and reversi games.

SEE ALSO

filedirs(info)

AUTHOR
NAME
abort - abort a script

SYNOPSIS
abort [arg ...]

DESCRIPTION
Abort provides a means to "abort" (abnormally terminate) a shell script. The
abnormal termination means the parent shell (in the case where a script
invokes a script) will terminate (as opposed to normal termination via 'exit'
where the parent script will continue with its next step).

Any arguments to abort are echoed to standard output. For example,

    if x$1 != x-I abort -unrecognized flag

would abort a script with the message "-unrecognized flag" if its first argu-
ment wasn't a "-l".

FILES
None

SEE ALSO
exit(tool), sh(tool)

AUTHOR
an Jacobson

BUGS/DEFICIENCIES
Abort should probably be implemented as an internal command to the shell,
not as a separate program. It should also work like the C-shell where abnor-
mal termination is indicated by a non-zero exit status which is set via an
optional parameter to "exit". E.g.,

    exit or exit(0) means normal termination
    exit(x) where x != 0 means abnormal termination
NAME
ac - assemble and link

SYNOPSIS
ac [-bcdlmotv] [-a arg] [-g arg] [-s arg]
    [program] [libraries...]

DESCRIPTION
Ac is a tool to assemble and optionally link assembly programs. The first
argument it encounters that is not a flag is assumed to be the name of the
assembly source. Any subsequent arguments are assumed to be names of
libraries. Ac uses the following flags.

-a the following argument is given to the assembler
-b just make a library
-c just make an object
-d make a debuggable output (binary, object, or library)
-g the following argument is given to the linker or librarian
-l make a listing file
-m make a loadmap
-o keep the object file
-s the following simple name is used for the name of the output files
    (binary, etc.)
-t do not link with the standard software tools library.
-v verbose mode. Rc will print all spawns to errout as they occur.

The b and c flags may not be used together. If neither -b nor -c is specified,
an executable file is written onto the bin subdirectory. The -b and -o flags are
also mutually exclusive.

The output files are written to specific directories with specific extensions.
The "name" below is the argument following the -s flag. If a name is given with
-s, it must be a simple name without extensions. If no name is given with -s,
the name defaults to the last simple name of the source file, without exten-
sions. If no source file name was given, the file name "acout" is used.

The output files are:
    library (-b):    lib/name
    object (-c):     lib/name
list (-l): src/name.lis
loadmap (-m): src/name.map
binary : bin/name

Ac recognizes the file name '-' to be the standard input for the source program, but not for the libraries. Also, if no program or libraries are given, ac will read from standard input.

Libraries specified will be expanded using the library search path, unless the library is specified with a full pathname.

FILES
+LIB/library - the standard tools library

SEE ALSO
fc(tool), ld(tool), rc(tool), comlb(lib)

AUTHOR
Marshall Spight

BUGS/DEFICIENCIES
NAME
admin - Create a TCS file

SYNOPSIS
admin -ifile file.tcs

DESCRIPTION
Admin 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(tool), get(tool), Unix admin(tool).

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

"- flag missing". The "-i" flag was not given.

"-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.

AUTHOR
Neil Groundwater at ADI.

BUGS/DEFICIENCIES
NAME
ar - archive file maintainer

SYNOPSIS
ar [-](d|p|t|u|x)[v] [-c[t]] arcname [files]

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 arcname [optional filenames]

where 'command' is any one of 'dptux', optionally concatenated with 'v', specifying what operation to perform on the archive file named 'arcname'. 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
The archiver will also contain the file's character count, type, and date and time of last change.

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.)

Normally, the archiver works by surrounding each file with a header and a trailer. However, for compatibility with the original archiver (by Kernighan-Plauger), command line options are available which allow the archiver to work by preceding a file by a header which contains its size: (The flags may appear anywhere on the command line.)

-ct Use character counts but also include the header/trailers. This will allow archives to work by using either character counts or header/trailers.

Binary files are allowed in the top level of an archive. However, they can NOT be used in nested archives.
NAME
args - use standard input as arguments for a tool

SYNOPSIS
args [shell flags] tool [arguments]

DESCRIPTION
Args invokes tool with any remaining arguments and the contents of standard input.

The most common use of args is as a form of argument explosion. For example, suppose the user wishes to delete all files which have the string "tst" somewhere in the filename. This may be accomplished with the following shell command line:

% Is find test | args rm

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 delete each file.

The first argument to args which does not start with a "-" is taken to be the name of the tool. A shell is spawned using dospwn and a command line composed of the tool name, all arguments following the tool name and the contents of standard input with newlines replaced with blanks.

The command line is passed to the shell as a string of characters and parsed by the shell in the usual manner. Thus, an unquoted line on standard input containing spaces may result in more than one argument. An unquoted line consisting only of spaces results in no argument.

If the user in the example above wanted to see the list of files passed to "rm", the following command line could be used:

% Is find test | args -v rm

The -v flag will cause the shell spawned by args to print the command lines passed to it.

FILES

SEE ALSO
sh(tool), dospwn(lib)
DIAGNOSTICS
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 shell is NOT spawned. This prevents the user from being destroyed by his/her indiscretion.

AUTHOR
Joe Sventek, modified by Michael O'Brien.

BUGS/DEFICIENCIES
If the arguments to be passed to "tool" on the args command line are to include io redirection, they must be escaped with an '@@' as in the following example:

% ls | find %a | args cat @@>alist

The redirections must be escaped, since args will open alist as it's standard output, rather than handing the string >alist to cat. You could also put the string in quotes, i.e., "cat >alist".
NAME
asplit - split corrupted archive file substituent files

SYNOPSIS
asplit [-vel] [-tstring] [fname...]

DESCRIPTION
Asplit reads the standard input and searches for lines that begin with the header flag for the archiver (#-h- starting in column 1), picks up the next non-blank string from the line (all characters are valid characters in a file name as far as asplit is concerned), and uses that as the name of an archived file. If that archived file is to be extracted from the corrupted archive (this decision is based on the fname argument - see below) a file by that name is created and all lines up to the next archive header are written to the file.

If -t is specified, the string following the -t is "tagged" onto the end of each file name before the file is created.

The -v flag causes the name of each file to be output onto ERROUT as it is opened.

The -e flag causes each line read by asplit to be echoed onto ERROUT.

The -l flag forces asplit to convert every file name to lower case letters. However, the tagged string, if any, is not affected.

If any arguments are specified which are not recognized as flags they are assumed to be strings to be used as filename patterns. If any fname arguments are specified, then asplit will proceed as follows:

1) Find 'the next' line beginning with a header flag.

2) Extract the archive file name.

3) Convert to lower case if requested.

4) If any of the fnames supplied is a sub-string of the archive file name (after case conversion, if any, but before the tag is appended, if any), the file is created and the appropriate lines are written to it. If there is no match, asplit proceeds to the next header flag.

By using the fnames argument a few select files can be split from a large corrupted archive file.

If no fnames are specified, all archive files are split from the corrupted archive.
FILES
None

SEE ALSO
ar(tool) - archive file maintainer

AUTHOR
Joe Sventec and Bob Upshaw

BUGS/DEFICIENCIES
If there is an attempt to create a file with a name larger than the system maximum (probably because of a large appended tag), asplit will print a message and abort.

There is a limit on the number of fnames which can be supplied. (The limit is currently 200 characters.)
NAME
back - back up directory trees (vax only)

SYNOPSIS
back [ -v ] [ -newtape ] [ -a <date> ]
  -v: verbose
  -newtape: backup tape is a newtape
  -a <date>: back up files created on/after 'date'

DESCRIPTION
'Back' backs up the specified files and directory trees onto tape, where they
can be recovered later, if necessary. The original files or directories are not
modified. Back is not an incremental backup tool (see backup(tool)).

The list of files and directories to back up is read from standard input. If a
directory name is specified, backup backs up everything in that directory,
everything in directories contained in that directory, and so on.

You must first allocate and mount the tape in DCL in order to use back:

  $ alloc mt
  $ mount/for mt

A list of all the files backed up is printed as the files are backed up. The -v
flag causes the dcl command lines given to the vax vms 'backup' utility to be
echoed by dcl as they are executed. The -newtape flag signals that the
backup tape has either never been used before, or it has just been initialized.
(Type 'dcl help initialize' to find how to initialize a tape.) The -a flag causes
files which have been modified on or since the date specified to be backed up.
Normally, all files specified are backed up. The date must be in standard
tools format:

  ddMmmyy[ hh:mm:ss[.s[s]]]]

I.e., the day must be two digits (supply leading zeros if necessary), the
month must be of the form 'Jan', 'Feb', 'Mar', etc., and the year must be two
characters long. The year 'yy' specifies the year 19yy. The remaining fields
are hours, minutes, and seconds. One or more spaces or tabs may be
between the year and the hour.

FILES
  /!mt - the tape drive back writes the files on
SEE ALSO
backup(tool), save(tool), restore(tool) Documentation for vax/vms BACKUP, ALLOC, and MOUNT

DIAGNOSTICS

back: tape is write locked. The tape must have a write ring to use back.

back: newtape flag specified and tape is not a new tape. The tape is NOT a newtape, so before executing back, you must re-initialize the tape using initialize in VMS. DO NOT use the newtape flag in this case.

back: tape is a questionable backup tape: please examine it. Inform the software resources group. The vax vms backup utility may have changed the format of its tapes.

back: tape is not a backup tape. The user is trying to back up files on a tape which isn't a backup tape. You can only backup files on tapes that have been used only for backups.

back: there are not an integral number of save sets on this tape. Either this tape is not a backup tape, or it has been garbaged.

back: error in backing up files. There was some problem trying to back up some files. For example, the user may not have had read permission for some of the files specified. The vax vms 'backup' utility should have given some descriptive error message.

back: vms error xxx(16). 'Back' was unable to do some tape operation. There may be something wrong with the tape. The error code is a vax vms error code.

AUTHOR
odd Hammond

BUGS/DEFICIENCIES
The listing of files backed up goes on the terminal rather than on standard output, since standard output is not currently inherited from parent to child.
NAME
backup - backup specified files/directories

SYNOPSIS
backup [-newtape] [file ...]

DESCRIPTION
Backup does an incremental backup for a user (i.e., saves everything on tape
that has changed since the last time a backup was done). The usual use of
backup is simply

    backup

which will save all of the files in the current working directory and all of its
sub-directories onto an existing tape. If the backup tape is a new tape, the
-newtape option should be used. Files or directories other than the current
working directory can be backed up. E.g.,

    backup a b/c

backs up everything in directories 'a' and 'b/c' that has changed since the
last backup, instead of the current working directory. (Since the date of the
last backup is kept on a file in the current working directory, use of file argu-
ments with "backup" can make the last backup date somewhat ambiguous).

The date of the last backup is kept on file "save.las" in the current working
directory (this means that you must have write permission in the current
working directory when you run backup). If there is no "save.las" file, backup assumes that no backup has ever been done and saves everything. If
there is a "save.las" file, backup appends the date of the backup onto the file,
maintaining a history of backup dates. Backup always uses the last date in
the file to figure out which files need to be backed up.

A list of all of the files in the directory(s) being backed up is put on the file
"file.str" which is backed up then removed. Having this file on the tape
allows the "recreate" script to reconstruct a user's file structure in the event
of a massive disk wipeout.

FILES
./save.las    for the date of the last backup
/dev/mt      to get to the tape drive (on the modcomp)
/mt          to get to the tape drive (on the vax)
./file.str   for the file structure being
             backed up.
./backup.tmp temporary file used by backup (on the vax)
./backdate.tmp temporary file used by backup (on the vax)
SEE ALSO
  back(tool), restore(tool)

DIAGNOSTICS

1. Modcomp

If the current working directory can't be written in, backup will blow up with the message

   "file.str: can't open"

It is strongly recommended that backup only be run from your login directory.

The message

   "-tape not in archive format"

usually means that the backup is going on a new tape and the "-newtape" flag wasn't used.

2. Vax

For a fairly complete description of error messages, see the manual entry back(tool).

A frequent error is that the tape is write-locked. In this case, you must dismount the tape, put a write ring in, mount the tape, and try your backup command again.

Sometimes the user will put a tape that is not a backup tape on and try to do a backup. Backup will print an appropriate message. If you want to save files on this tape, and loose all the information which was on it before, you must initialize the tape. To do this, put your tape on the tape drive, get in dcl, and say:

   $ allocate mt
   $ initialize mt
   $ label: backup

You will then need to de-allocate the tape before using the backup tool. Remember that this is not the same as -newtape. Also see the dcl 'initialize' command.
AUTHOR
Van Jacobson on the modcomp, Todd Hammond on the vax.

BUGS/DEFICIENCIES
It is strongly recommended that you do NOT allow a tape to become full (where full means that the entire tape holds meaningful data). On the modcomp, the tape will go off the reel, and it’s hard to get back on. On the vax, if you fill a tape while backing up, the operator console in the computer room will beep and say that there is a request by a user to mount a new volume on the tape drive. You will then have to put a new tape on the tape drive, and whenever you want to recover from those tapes, you will have to put first one tape on the drive, then the other, then the first again, and finally the second one, which is a big pain.
NAME
cat - concatenate and print text files

SYNOPSIS
cat [file1] [file2]...

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.

FILES
None

SEE ALSO
The UNIX tools cat, pr, cp

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

AUTHOR
B. Kernighan and P.S. Plauger

BUGS/DEFICIENCIES
Using the same file for output as well as input is not allowed.
NAME
cd - change working directory

SYNOPSIS
cd directory

DESCRIPTION
Cd makes "directory" the new working directory. When the user makes
references in the future to files that are not full pathnames, the files area
assumed to be in "directory".

Cd is executed directly by the shell, so if the user types "cd", the shell will
execute this procedure itself, even if the user has a program by the name
"cd" in his search path.

Cd only changes the current directory of the current process. Future
processes spawned from the current process will have the same working
directory as the parent process, but cd will not change the working directory
of a parent process.

SEE ALSO
pwd(tool), files(info)

AUTHOR
Bob Upshaw

BUGS/DEFICIENCIES
On the vax it is not possible for the shell to affect the logical name used to
keep track of the current disk drive; as a result, if a user enters a shell from
dcl, changes his working directory, and exits the shell, his current working
directory is not defined (i.e. his disk drive may be incorrect.)
NAME
ch - make changes in text files

SYNOPSIS
ch [-ax] [expr ...] 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 "expr" 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///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 substi­
tution. 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 informa­
tion.) In particular, if n=3,
ch expr from to
is equivalent to the ed command
g/expr/s/from/to/g
while
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.
If "toexpr" is unspecified (or null), the matched string is deleted. If a dele­
tion is desired with the multiple leading tag expressions, a "toexpr" of "" -i.e.
quotes around an empty string may be used.

There are two metacharacters allowed in the substitution string: the "ditto"
character, &, stands for whatever was matched by "fromexpr" and is used for
inserting into lines. The substring indicator, @n where 0<=n<=9, stands for
n'th matched substring (substrings are things between @(...) pairs -- see
Substrings are generally used for rearranging fields of input lines.

A text pattern consists of the following elements:

- @tc@tliteral character
- @t?@tany character except newline
- @t%@tbeginning of line
- @t$@tend of line (null string before newline)
- @t[@...@]@tcharacter class (any one of these characters)
- @t![...@tnegated character class (all but these characters)
- @t*@tclosure (zero or more occurrences of previous pattern)
- @t@c@tescaped character (e.g., @%, @[, @*)

Any part of a text pattern may be enclosed by " @( ... @)" pairs. The "@(" and "@)" are ignored by the pattern matcher but the text matched by their part of the pattern is saved and may be used in later substitutions via the "@n" (where n is a digit between 0 and 9) construct.

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

- @t%@tnot at beginning
- @t$@tnot at end
- @t*@tat beginning

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

- @tc@tliteral character
- @ta-b@trange of characters (digits, lower or upper case)
- @t!@tnegated character class if at beginning
- @t@c@tescaped character (@! @- @@ @

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

- @t!@tnot at beginning
- @t-@tat beginning or end

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

- @tn@tnewline
- @ta-t@ttab
- @tafformfeed
- @t@0 to @9@tsubstring insertion
- @t@c@tc (including@@).
Care should be taken when using the characters % $ [ ] ! * @ and any shell characters in the text pattern. It is usually necessary to enclose the entire substitution pattern in quotes.

**An example:** Say you wanted to change the order of the two parameters to subroutine "gronk" throughout a large program. The ch command:

```
ch "%@*(call gronk *@(@)@(@(??)),@@(@(??))$* @@2,@@1)"
```

would probably do it. The part of the pattern that makes up the first substring picks any lines of the form:

```
"call gronk"
"    call gronk"
"    call gronk"
```

The second substring picks up everything to the comma which separates the two arguments and the third substring picks up everything from the character after the comma up to the closing paren. This will match lines of the form

```
call gronk ( a, b )
call gronk ( (25+q/15)**11, f(b)-3 )
```

and produce

```
call gronk ( b, a )
call gronk ( f(b)-3, (25+q/15)**11 )
```

(but note that it will be fooled by lines of the form

```
call gronk ( a, f(b,c) )
```

and produce

```
call gronk ( c, a, f(b )
```

which helps to point out that one should carefully inspect the output produced by any non-trivial change expression.

**FILES**

None

**SEE ALSO**

ed(tool), find(tool) The UNIX tool GRES "Software Tools", pgs 135-154
DIAGNOSTICS

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

AUTHOR

'ch' was originally implemented on BKY by Debbie Scherrer from Kernighan and Plauger's "Software Tools". Major modifications were performed by Joe Sventek. Substrings [@(...)@] were added by Van Jacobson.

BUGS/DEFICIENCIES

A minus sign(-) may not start an expression. It is also possible for the shell to pass an argument to ch that is too long for the tool to handle. In that event, the argument is truncated without warning.
NAME
changes - show changes in the on-line documentation

SYNOPSIS
changes [<ndays>]

DESCRIPTION
Changes shows changes in the on-line documentation (the Manual and News).
If the argument <ndays> is present, documents which have changed in the
last <ndays> are listed. E.g.,

changes 7

lists all documents which have changed in the last week.

If there is no argument, 'changes' will list documents which have changed
since the last time you ran 'changes' (the last date you ran changes is kept
on the file 'last.cha')

If there is no argument and you have no 'last.cha' file, changes in the last 30
days are listed.

FILES
/usr/man/*
/etc/helpstuf/*
'~/last.cha'

SEE ALSO
help(tool),man(tool)

AUTHOR
Van Jacobson
NAME
   chmod - change file access protections

SYNOPSIS
   chmod [-s[rwed]] [-o[rwed]] [-g[rwed]] [-w[rwed]] file ...

DESCRIPTION
   Chmod changes the file protections of the specified file(s). There are four
   subfields, system, owner, group and world. They may be changed by using the
   following flags:

   -s Change the system file protections.
   -o Change the owner file protections.
   -g Change the group file protections.
   -w Change the world file protections.

   Each subfield flag may be followed by some combination of the letters
   "rwed", to give the specified access to the file. The letters "rwed" stand for
   read, write, execute and delete.

   Subfields that are not specified are not changed.

   If chmod is used without any flags, the file protections of the specified file
   are not altered.

   Examples:

   chmod -wre file

   Changes the world access of "file" to read and execute. System, owner and
   group access is not affected.

   chmod -orwed -g -w file

   Gives the owner all access and denies all access to the group and world. Sys-
   tem access is not changed.

   chmod -g -w bin/

   Removes all group and world access for the directory "bin."
SEE ALSO
ln(tool)

DIAGNOSTICS

"chmod: Error getting file protections - file"
This error occurs when chmod cannot open the specified file for reading.
Possibly the file does not exist.

"chmod: Error getting file protections - file"
This error will occur when changing the file protection causes a privilege vi­

Author
Craig Leres

BUGS/DEFICIENCIES
Chmod needs write access to the file in order to change the protections.
Thus, if you protect a file against yourself, chmod will not be able to unpro­
tect it. However, the DCL command SET PROTECTION will still work.

Since, on the vax, it is possible to have a directory and file with the same
name it is suggested that ".dir" be added to directory names and "." be
added to normal file names.

For example, if the current directory contains a file called "src" and also a
directory called "src", it is suggested that "src.dir" be used to change the
protections of the directory and "src." be used to change the protections of
the file.
NAME

cleanmtree - Cleanup subtree of file structure

SYNOPSIS

cleanmtree [dir]

DESCRIPTION

Cleanmtree uses the "cleanup" tool to clean up an entire subtree of the file structure. If an argument is given, cleanmtree does a "cleanup" of that directory and all of its sub-directories. If no argument is given, cleanmtree works on the current working directory and its sub-directories. "Clean up" means that all files that are not the most recent version are deleted, and all files after this deletion that do not have version numbers of '1' are renamed to have version numbers of '1'.

SEE ALSO

cleanup(tool), istree(tool). The DCL commands $PURGE and $RENAME.

DIAGNOSTICS

$PURGE-I-NOFILPURG, no files purged for <dec file name>.
This message is given from the DCL $PURGE command; it simply indicates that the given directory had no files names with multiple version numbers.

AUTHOR

Vern Paxson
NAME
  cleanup - Clean up files in directory

SYNOPSIS
  cleanup [dir..]

DESCRIPTION
  Cleanup is a tool that cleans up files in the specified directories. If no directories are specified, cleanup cleans up the working directory. "Cleans up" means that all files that are not the most recent version are deleted, and all files after this deletion that do not have version numbers of '1' are renamed to have version numbers of '1'. For example, if directory /mnt/dirty contains

  garbage
  junk.dmp.139
  mess
  mess..2
  mess..3
  trash

then the command "cleanup /mnt/dirty" would alter the directory so it looks like:

  garbage
  junk.dmp
  mess
  trash

Since the filename '-' is interpreted as standard input, a 'cleantree' script can be written which cleans up an entire sub-tree of the file structure:

  lstree -dp $1 | find /@$ | cleanup -

SEE ALSO
  cleantree(tool). The DCL commands $PURGE and $RENAME.

DIAGNOSTICS
  %PURGE-I-NOFILPURG, no files purged for <dec file name>.
  This message is given from the DCL $PURGE command; it simply indicates that the given directory had no files names with multiple version numbers.

AUTHOR
  Vern Paxson
NAME
cmp - compare two files

SYNOPSIS
cmp file1 file2

DESCRIPTION
File1 is compared line-by-line with file2. If any lines differ, cmp announces
the line number and prints each file’s offending line.

FILES
None

SEE ALSO
comm(tool), diff(tool)
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.

AUTHOR
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] [-i] 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.

   The optional argument -i tells comm to ignore case - "A" and "a" are treated
   the same.

FILES
   None

SEE ALSO
   cmp(tool), UNIX diff

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

AUTHOR
   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
confirm - get confirmation from user

SYNOPSIS
confirm <message>

DESCRIPTION
confirm prints <message> on the user's terminal, followed by a question mark. It then reads the next line typed by the user. If the first letter on this line is y or Y, confirm ends peacefully. If the first letter is n or N, confirm aborts. If the character read is not y, Y, n, or N, confirm prompts the user with y/n? until it receives a y/n response.

For instance, if confirm appeared in a script containing the following:

    ... confirm continue ...

then the line

confirm continue?

would be printed on the user's terminal. If the user typed 'y' or 'Y', execution of the script would continue.

Note that confirm always writes to and reads from the user's terminal.

SEE ALSO
    echo (tool), UNIX confirm (tool)

AUTHOR
Anne Herrmann

BUGS/DEFICIENCIES
Since the maximum size of a message is machine dependent, confirm should only be used with a message of 80 characters or less. To "confirm" larger messages, "echo" them with the echo tool and then use confirm.
NAME
   cp - copy files

SYNOPSIS
   cp [-dv] file ... dest_file

DESCRIPTION
   The first files(s) are copied onto dest_file.

   If dest_file is a directory, each file is copied into dest_file using its last simple
   name as the filename. E.g.,

   cp /mnt/joey/who scripts

   where scripts is a directory, would copy /joey/who onto scripts/who. The
   special name ' ' for 'dest_file' refers to the current working directory.

   If dest_file isn't a directory or doesn't exist, the file(s) are simply copied onto
   file 'dest_file'.

   If the destination file exists, cp will query the user with the message

   cp: File <dest_file> already exists; replace?

   and wait for a y/n response before continuing (this query is always directed
   to the user's terminal, even if cp is being used in a script). The question can
   be defeated by using the -d (destroy) flag, in which case cp will copy without
   checking whether the destination file exists.

   The -v induces verbosity; if it is specified, cp will echo information about what
   it is doing.

SEE ALSO
   cat(tool), pr(tool), mv(tool) Unix cp(tool), cpall(tool)

AUTHOR
   Bob Upshaw

BUGS/DEFICIENCIES
   Some confusion may occur if multiple versions of a file exist. See the manual
   entry for mv(tool).

   Cp has special coding to get around a bug with DCL COPY (which cp uses).
   This bug manifests itself by NOT correctly updating the modification date of
   the destination file if the destination file name is not fully specified (e.g. a
   directory was given.) Cp will always attempt to supply the exact destination
   file name in this case. However, if it can't (for example, in the case of multi­
   ple object file names supplied), the modification date of the destination
   file(s) will be incorrect. This can be devastating for normal backup pro­
   cedures (see backup(tool)).
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

SEE ALSO
expand(tool)

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

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

BUGS/DEFICIENCIES
NAME
cptree - copy directory and all files contained

SYNOPSIS

cptree [-d] dir1 dir2

DESCRIPTION

Cptree copies the specified source directory and all of the files and subdirectories it contains to the specified target directory, creating new directories if necessary. If cptree finds a file that conflicts between the source and target directories, it prompts the user with the message:

    cp: File <file> already exists; replace?

A response of "y" or "Y" gives cptree permission to overwrite this file; any other response causes cptree skip this file and continue. Files in the target directory that do not conflict with files in the source directory are not be modified.

Cptree accepts one flag:

-d No questions are asked, even if files in the source directory conflict with files in the target directory. Files in the target directory may be updated with files from the source directory.

SEE ALSO

cp(tool), mkdir(tool), lstree(tool), rmdir(tool)

AUTHOR

Craig Leres
NAME
crt - copy files to terminal

SYNOPSIS
crt [-n] [files...]

DESCRIPTION
Crt is similar to 'cat' except that it prints only n lines (default is the screen size of the terminal, minus 3 lines) at a time. After each set of lines are printed, crt will wait for instructions from the user. Hitting a carriage-return will cause the next n lines to appear, hitting a 'q' followed by a carriage return will cause crt to quit printing the current input file, and hitting an end-of-file character 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.

SEE ALSO
cat(tool)

DIAGNOSTICS
Crt will default to 24 line screen size and complain if it can't find the size of the terminal.

AUTHOR
Marshall Spight
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

SEE ALSO

DIAGNOSTICS

AUTHOR
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
date - print the current date

SYNOPSIS
date

DESCRIPTION
The current date and time are printed on standard output in the STANDARD format.

SEE ALSO
getnow(lib), fmtdat(lib)

AUTHOR
Marshall Spight
NAME
dc - desk calculator

SYNOPSIS
dc [files ...]

DESCRIPTION
Dc evaluates arithmetic expressions from the source files, an arbitrary number of expressions separated by semicolons or newlines. If no input files are given, or the filename '-' is specified, dc reads from the standard input.

Ordinarily dc operates on decimal arithmetic expressions, but the user may specify an input base and output base other than decimal (but real numbers are converted to integers in bases other than 10, e.g., '1.5' will print as '1' in base 16). All numbers must begin with an integer, no matter what the base.

An expression is a normal arithmetic expression using numbers, variables, parentheses, operators, and functions. Below is a list of the legal dc functions, followed by the list of legal operators listed in order of precedence:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>min(a1, a2, ..., an)</td>
<td>minimum value of a1, a2, ..., an.</td>
</tr>
<tr>
<td>max(a1, a2, ..., an)</td>
<td>maximum value of a1, a2, ..., an.</td>
</tr>
<tr>
<td>abs(a)</td>
<td>absolute value of 'a'</td>
</tr>
<tr>
<td>log(a)</td>
<td>logarithm of 'a'</td>
</tr>
<tr>
<td>ln(a)</td>
<td>natural logarithm of 'a'</td>
</tr>
<tr>
<td>sin(a)</td>
<td>sine of 'a'</td>
</tr>
<tr>
<td>cos(a)</td>
<td>cosine of 'a'</td>
</tr>
<tr>
<td>tan(a)</td>
<td>tangent of 'a'</td>
</tr>
<tr>
<td>exp(a)</td>
<td>exponential of a</td>
</tr>
<tr>
<td>sqrt(a)</td>
<td>square root of a</td>
</tr>
<tr>
<td>asin(a)</td>
<td>arcsine of 'a'</td>
</tr>
<tr>
<td>acos(a)</td>
<td>arccosine of 'a'</td>
</tr>
<tr>
<td>atan(a)</td>
<td>arctangent of 'a'</td>
</tr>
</tbody>
</table>
All binary operators except exponentiation are left associative. Exponentiation, assignment and conditional evaluation are right associative. The relational operators are non-associative.

An assignment looks like:

    name=expression

where 'name' is a character string of (virtually) any length, starting with a letter and consisting of only letters and digits. Variables are automatically declared and have the value 0 until they appear on the left of an assignment. Variable names are not case sensitive (e.g., 'x' and 'X' are the same variable).
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). For example,

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

would accept input in binary and print results in hexadecimal.

(2) Dc will allow ibase and obase to be set to real numbers, but they will be truncated. For example,

```
ibase = 2.3
ibase will be truncated to 2.
```

(3) 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).

Some examples of the dc expressions are:

```
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 + 0f
```

would print the answer "101001"

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

would print the answer "356"

```
( num1 < num2 ) ? num1, num2
```

would print the minimum of "num1" and "num2".
A script to do a numeric integration of a file of data kept as 1 real value per line could be:

```
ch "^*" 'sum = sum + &' <dat | dc >int
```

**FILES**
None.

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

**DIAGNOSTICS**

*assignment error* - means the thing of the left of an '=' wasn't a name.

*Tried to associate non-associating operator* - means an attempt was made to associate a relational operator.

*expression stack overflow* - arithmetic expressions have been nested too deeply.

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

**AUTHOR**
Philip H. Scherrer (Stanford U.)
Van Jacobson added real number arithmetic, the 'round' operator, changed & | and ! to be boolean rather than logical, rewrote the assignment handling and changed expression processor to allow multiple expressions per line. Theresa Breckon converted the source to yacc input form and added the conditional operator and the functions.

**BUGS/DEFICIENCIES**
Dc won't recognize a number of the form '.dd' (e.g., '.09') so you must either put a leading zero (e.g., 0.09) or use scientific notation (e.g., 9e-2).

When the output radix is not 10, numbers larger than the host machine's default integer size will not be output correctly.
NAME
dcl - execute DCL commands

SYNOPSIS
dcl [-lvn] [+command ... [>output]]

or

dcl [-lvn] <DCL_commands [>output]]

DESCRIPTION
Dcl executes its arguments as DCL commands (where "dcl" is the tool dcl and "DCL" is the command line interpreter on the VAX). If more than one argument is given, dcl considers them to be arguments to the DCL command:

% dcl show terminal

This is the same as the $SHOW TERMINAL command in DCL. More than one command may be put on one line by separating them with a plus sign: For example:

% dcl allocate mta0: +mount/for/den=800 mta0: example

Commands may also be quoted:

% dcl "allocate mta0:" +"mount/for/den=800 mta0: example"

Note that the two previous examples are exactly the same. The above two features may also be combined:

% dcl "show logical" +assign dra0: mydisk:

The space before the plus sign is necessary. If there are no arguments, dcl will read its commands from standard input. The output of the commands executed is put on standard output.

The following flags are implemented:

-1 Execute the login file ~/login.com before executing the commands.

-v Verbose: execute the DCL command $set verify before executing the commands. This will cause dcl to write every command it executes to the terminal before executing it.

-n Don't abort if dcl or DCL gets errors. This flag automatically gives the command $set noon. In addition, the tool dcl will not abort either if this flag is specified.
If the -l flag is specified, the verbose option is in effect only after the
"/login.com file is executed.

DCL will abort if there are any errors in the commands executed, unless the
user gives the command $set noon in groups of commands where he doesn't
want DCL to abort or specifies the -n flag. In this case, the user should set
the exit status when these groups of commands are finished with the $exit
command.

SEE ALSO
sh(too) (the tools command interpreter) type "dcl help" for information on
dcl commands available.

DIAGNOSTICS
dcl: error in executing DCL command(s)

Either dcl couldn't be executed, or there was a fatal error in executing
one of the DCL commands specified.

AUTHOR
Todd Hammond

BUGS/DEFICIENCIES
If dcl is to read from standard input (either from a file or from a here docu-
ment), the DCL commands must be preceded by a dollarsign ($). Further-
more, if dcl is reading from a here document, the dollarsign must be escaped
(@$).
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). If neither a fieldlist nor a delimiting character is supplied, '-t' with a tab is assumed.

Once fields have been described 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. (The argument $0 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

The normal escapes are allowed in the output format (e.g., '@n' for newline, '@t' for tab, 'f' for formfeed). If a '$' is to appear in the output, it must be escaped with a '@'. 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

`sed` (tool)

AUTHOR

David Hanson and friends (U. of Arizona)

BUGS/DEFICIENCIES

The choice of `$` for the argument indicator is somewhat unfortunate since it is also used by the shell. Remember that the shell will do parameter substitution on arguments in double quotes so it's wise to get in the habit of enclosing the 'outputformat' in single quotes. Because input fields can only be manipulated by specifying arguments in the output format, the output format is assumed to have references to such arguments.
NAME
find - search a file for text patterns

SYNOPSIS
find [-acfx] expr [expr ...]

DESCRIPTION
Find searches the standard input file for lines matching the text patterns "expr" (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. And finally, if the -f option is specified, lines that contain the expression in any case are considered as matching.

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)
- @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

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:

@n newline
@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(tool), ed(tool), ch(tool) and the UNIX grep command.

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

AUTHOR
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
finddate - search a file for "matching" dates

SYNOPSIS
finddate [-p position] [-d date] [-c count]

DESCRIPTION
Finddate searches the standard input file for lines which "match" a user specified date. All matching lines are copied to standard output.

The -p flag specifies at which column position in each line the date starts. If no -p flag appears, '-p 1' is assumed. The date is expected to be in one of the forms:

ddmmmyy (e.g., 06may80)
dd-mmmy-yy (e.g., 06-may-80)
mm/dd/yy (e.g., 05/06/80)

Leading blanks are ignored and letters may be in either case. If no date is found at the specified position, the line is ignored.

The date to match against is specified via one of either the -d or -c flags:

-d specifies a date (e.g., -d 01apr80, -d +15jan76, -d -02feb79).
-c specifies a count of days before the current date (e.g., -c 30, -c -7, -c +90).

The sign of the date or count argument specifies the matching criteria: "+" matches any date earlier than the argument, "-" matches any date later than the argument, and no sign matches any date equal to the argument (e.g., -d +4May80 matches all dates before 04May80, -d -4May80 matches all dates after 04May80 and -d 4May80 matches only 04May80.

Some examples of finddate's use are:

lstree -pl | finddate -c -7 \ ch "%?*" \ save
back up all local files that have changed in the last 7 days.

lstree -pl | finddate -c +60 \ ch "%?*" "rm" \ sh
deletes all files that haven't been referenced in the last 2 months.

lstape -l | finddate -p 39 -d -15apr80
list all files put on an archive tape after 15apr80.
SEE ALSO

is(tool), lstape(tool), lstree(tool), save(tool) The Unix tool 'find'.

DIAGNOSTICS

None.

AUTHOR

an Jacobson

BUGS/DEFICIENCIES

The +/- convention for before/after is obscure, to say the least: It is the convention used by Unix 'find' and was chosen for compatibility.

Dates earlier than Jan 1, 1961 or later than Dec 31, 1999 won't be recognized (due to limitations of 16 bit integers).
NAME
finger - list information about current users

SYNOPSIS
finger

DESCRIPTION
Finger provides information about the users currently logged into the system. It shows terminal name, user name, length of time the terminal has been idle in seconds, and the real name of the user. Entries are sorted by terminal name.

Sample output:

<table>
<thead>
<tr>
<th>Terminal</th>
<th>User</th>
<th>Idle Time</th>
<th>Real Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>opa0</td>
<td>system</td>
<td>57:32</td>
<td>System Manager</td>
</tr>
<tr>
<td>tta1</td>
<td>bobup</td>
<td>26 Bob Upshaw</td>
<td></td>
</tr>
<tr>
<td>tta7</td>
<td>leres</td>
<td>2 Craig Leres</td>
<td></td>
</tr>
<tr>
<td>ttb0</td>
<td>superman</td>
<td></td>
<td>Clarke Kent</td>
</tr>
<tr>
<td>ttb2</td>
<td>cool</td>
<td>12 Joe Cool</td>
<td></td>
</tr>
<tr>
<td>ttb3</td>
<td>barale</td>
<td>1:01</td>
<td>Paul Barale</td>
</tr>
<tr>
<td>ttc7</td>
<td>harold</td>
<td>7 Harold of the Arrow</td>
<td></td>
</tr>
</tbody>
</table>

In this example, the System Manager has been idle for fifty seven minutes and thirty two seconds. User superman is not idle.

SEE ALSO
who(tool), ps(tool)

DIAGNOSTICS

"finger: Not installed with WORLD."
"finger: Not installed with CMKRNLD."

These error messages occur when finger is not installed properly. In order for non-privileged users to run finger, it must be installed with both WORLD and CMKRNLD privileges. When one of these errors occur, complain to the system manager.

"finger: idlib$idle(), <vms error text>"
"finger: sys$getjpi(), <vms error text>"

This should never happen. If it does, please report it to the system manager.

AUTHOR
Craig Leres
NAME
form - produces form letter by prompting user

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 a line of input from the user and replaces the brackets and string with what the user typed.

Normally only one line of input is accepted from the user. However, the user can continue his response on succeeding lines by terminating each line to be continued with an atsign ("@").

The prompts inside the file may also span line boundaries if so desired (no atsign is needed in this case).

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

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 '>'.

FILES
The user's terminal is opened both at READ and at WRITE 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 the either the prompt or the response is too long for the tool's internal buffer.
NAME
fyi - provides system information from keywords

SYNOPSIS
fyi [-d dir] [-<sec>] [-a] keyword ....

DESCRIPTION
Fyi is a "for your information" tool which gives information about the system. Fyi looks at manual entries in the specified directories which pertain to the "keyword" specified. These keywords can be any regular expression accepted by the find tool. The manual entry name and a one line explanation are printed for entries that match any of the keywords. For example:

% fyi print
  cat - concatenate and print the text
  comm - print lines common to two files
  lpr - queue a file to the printer
  pl - print specified lines/pages in a file
  pr - paginate files for printing
  tail - print last lines of a file

By default, fyi will give information from /usr/man/tool. Fyi understands the following flags:

-d specifies from which top-level directory to obtain the information.

-sec specifies the manual section name from which to get the information. For example, '-lib' will get information about primitives from "man/lib" and '-tut' will give information from "man/tutorial". To get information about printing from the lib section of /misc for example:

% fyi -d /misc -lib print

-a says to search for lines that match ALL of the keywords specified on the command line. For example:

% fyi -a print lines

would give the following:

comm - print lines common to two files.
pl - print specified lines/pages in a file.
tail - print last lines of a file.
FILES
   /.../man/<sec>.f - contains manual titles from /.../man/<sec>tion/*

SEE ALSO
   find

AUTHOR
   DePriest Dockins

BUGS/DEFICIENCIES
   Because children processes don't inherit the standard outputs of parents
   and fyi spawns to another tool, you cannot redirect the standard output of
   the fyi tool.
NAME
get - get generation from TCS file

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

DESCRIPTION
Get retrieves earlier versions of text from "historyfile" as computed by
delta(tool). The possible flags are:

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

-h Write the history information associated with the versions to the
standard output. 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 into the "result" file.

The retrieved version will be put in the "result" file if specified, otherwise it is
sent to the standard output. History information is sent to the error output.

SEE ALSO
admin(tool), delta(tool), Unix get(tool).

DIAGNOSTICS
"usage: get [-h][-rM.N] historyfile [resultfile]". Correct calling format is pro-
vided 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(tool) 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.

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

"Cannot locate TCS history file." Could not find file supplied for historyfile.
AUTHOR
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
NAME
goto - sh command transfer

SYNOPSIS
goto label

DESCRIPTION
Goto is allowed only when the sh is taking commands from a file. The file is
searched from the beginning for a line beginning with a ':' followed by one or
more blanks followed by the label (one or more non-blank characters). If
such a line is found, the goto command returns, leaving the command file
positioned such that the next line read will be the line immediately following
the label.

' :' is a do-nothing command that is ignored by the sh and only serves to place
a label.

SEE ALSO
sh(tool), if(tool), exit(tool)
The UNIX command goto.

AUTHOR
Bob Upshaw

BUGS/DEFICIENCIES
Because of the nature of the sh, goto is implemented as a sh command.

The maximum length of a label is 30 characters. Any more supplied and the
goto statement will be flagged as a syntax error.

The label on the goto statement must be followed by a blank or end of line.
Any other junk characters after the blank are ignored.
NAME
head - copy first lines of files to output

SYNOPSIS
head [-n] [file ...]

DESCRIPTION
Head is similar to 'cat' except that it prints only the first n lines (default screensize).

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.

FILES

SEE ALSO
cat(tool), crt(tool)

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

AUTHOR
S. J. Mellor

BUGS/DEFICIENCIES
NAME
help - more on-line documentation

SYNOPSIS
help [subject]

DESCRIPTION
Help supplies on-line consultation regarding the specified subject. If no subject is supplied, help gives an indication of what items are available.

FILES
/etc/helpstuf/* - help articles

SEE ALSO
man(tool)

DIAGNOSTICS
A warning is printed if the subject is not known to help.

AUTHOR
ob Upshaw

BUGS/DEFICIENCIES
none
NAME
  if - conditional sh command

SYNOPSIS
  if ( expr ) command [ arg ... ]

DESCRIPTION
  if evaluates the expression expr, and if its value is true, executes the given
  command with the given arguments.

  The following primitives are used to construct the expr:

  -r file  true if the file exists and is readable
  -w file  true if the file exists and is writeable
  -x file  true if the file exists and is executable
  s1 = s2  true if the strings s1 and s2 are equal
  s1 != s2  true if the strings s1 and s2 are not equal
  s1 gt s2  true if the string s1 is lexicographically
             greater than (using the ASCII collating
             sequence) s2

  These primaries may be preceded by the following unary operators
  (separated with one or more blanks or tabs):

  !      unary negation operator

  The command that the shell executes if the conditional is true may be any
  command that the shell could execute between semicolons. Thus, "if (expr) a
  | b" groups as "if (expr) (a|b)".

  Since the syntax for programs and if statements is different, it is possible to
  have a tool or script named "if" without getting the shell confused.

SEE ALSO
  sh(tool), goto(tool), exit(tool)

DIAGNOSTICS

AUTHOR
  Marshall Spight

BUGS/DEFICIENCIES
  It is quite common to compare, say, argument 1 ($1) with a string. However,
  the following will not work if $1 is empty:

  if ( $1 = dog ) echo arf arf

  since after argument processing this gets turned into:
if (tool)

if ( = dog ) echo arf arf

To fix this, say instead:

if (x$1 = xdog) echo arf arf
NAME
includ - expand included files

SYNOPSIS
includ [file] ...

DESCRIPTION
Includ copies the input file(s) to the standard output. Whenever an input line begins with include filename the entire contents of filename will be copied to standard output in place of the include line. If includ is invoked with no arguments or one of the arguments is "-", it reads standard input.

Filename may be enclosed in double quotes. Filename must be enclosed in double quotes if non-alphanumeric characters are used.

Includes may be nested (i.e., a file being "included" may contain its own includes). The depth of included files allowed is machine dependent.

If the filename on an "include" line is not a full pathname, the standard include path, +INCL, is prepended to the filename before the attempt to open the file is made. All files used by includ are opened using "open". Note that the standard include path is NOT searched when includ is opening files given on the argument line, just files specified in 'include' statements.

SEE ALSO
paths(info), open(lib) Kernighan and Plauger's "Software Tools", pages 74-77.

DIAGNOSTICS
includes nested too deeply - exceeded maximum number of open files allowed.

t open - . ta 150uR
filename: can't open - File could not be located or maximum number of opened files was exceeded.

AUTHOR
Original code by Kernighan and Plauger in "Software Tools"
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 markl. 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 markl is then output as decimal integers before the index record is flushed.
NAME
kill - kill a process

SYNOPSIS
kill pid ...

DESCRIPTION
Kill kills the processes specified by the process id's.

Normally, you are only allowed to kill processes that are in your uic. Privileged users (who have SETPRV or WORLD privilege) can kill any process.

SEE ALSO
ps(tool)

DIAGNOSTICS
A message is output if the specified process does not exist or if it isn't possible to kill the process.

AUTHOR
Craig Leres

BUGS/DEFICIENCIES
NAME
kwic - make keyword in context index

SYNOPSIS
kwic [-t throwout_file ...] [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 origi­
nal 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.

It is usually desirable to delete uninteresting words from the kwic output
(e.g., words like 'a', 'the', 'of', etc.). "Throwout lists" may be given to kwic via
the '-t' flag. This flag should be followed by the name of a file containing the
words to throw out. Kwic will not output a line starting with words in the
throwout list. The format of a throwout file is simply a list of words
separated by blanks, tabs or newlines. Blank lines are ignored and com­
ments (starting with a '#') are allowed anywhere in the file.

FILES
/etc/kwicto (kwic ThrowOut) a throwout list of common words

SEE ALSO
unrot(tool), sort(tool)

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

AUTHOR
Original from Kernighan and Plauger's 'Software Tools', with modifications by
Debbie Scherrer. Throwout lists added by Van Jacobson.

BUGS/DEFICIENCIES
NAME
lam - laminate files

SYNOPSIS
lam [-string] (files -string)....

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: fool "-, file2:" fo02

results in output lines that look like

file1: a line from foo1, file2: a line from fo02

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(tool), tail(tool)

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.

AUTHOR
David Hanson and friends (U. of Arizona)

BUGS/DEFICIENCIES
NAME
ld - link and produce binary

SYNOPSIS
ld [-dmtv] [-g arg] [-s arg] [program] [libraries...]

DESCRIPTION
ld is a tool to link programs. The first argument it encounters that is not a flag is assumed to be the name of the object file. Any subsequent arguments are assumed to be names of libraries. ld uses the following flags.

-d make a debuggable output (binary, object, or library)
-g the following argument is given to the linker or librarian
-m make a loadmap
-s the following simple name is used for the name of the output files (binary, etc.)
-t do not link with the standard software tools library.
-v verbose mode. ld will print all spawns to errout as they occur.

An executable file is written onto the bin subdirectory.

The output files are written to specific directories with specific extensions. The "name" below is the argument following the -s flag. If a name is given with -s, it must be a simple name without extensions. If no name is given with -s, the name defaults to the last simple name of the source file, without extensions. If no source file name was given, the file name "ldout" is used.

The output files are:

    loadmap (-m):    src/name.map
    binary           bin/name

ld recognizes the file name '-' to be the standard input for the source program, but not for the libraries.

Libraries specified will be expanded using the library search path, unless the library is specified with a full pathname.
FILES
  +LIB/library - the standard tools library

SEE ALSO
  rc(tool), fc(tool), comlib(lib)

AUTHOR
  Marshall Spight

BUGS/DEFICIENCIES
  Since fcopy cannot handle object files, ld is not actually able to read its input from stdin.
NAME
lex - lexical analyzer generator

SYNOPSIS
lex @[-distv@] @[file@]
   -d: lexical analyzer should operate in Debug mode
   -i: case Insensitive lexical analyzer
   -s: Suppress default rule
   -t: Trace processing
   -v: Verbose

DESCRIPTION
lex generates a program from its input file to perform lexical analysis of text. Such a program is often called a scanner. The input file to lex (if none specified, the standard input it used) contains regular expressions for which the scanner should search its input text. The input file also contains actions (ratfor program fragments) which will be executed when their corresponding regular expression is matched. The output from lex is ratfor code defining the integer function yylex. When a program containing yylex is compiled, linked, and executed, it will search its input text for strings matching the regular expressions, and execute the corresponding actions. Text that is not matched by any expression is simply copied to the output.

Here is a sample lex program:

```ratfor
%%
[ @t]+$ [ @t]+ call putc( ' ' )
[ @t]*@n
%%

 DRIVER
   integer yylex, dummy
   dummy = yylex( 0 )
   DRETURN
end
```

This program removes all trailing whitespace (blanks or tabs), compresses multiple whitespace characters into a single blank, and deletes lines which contain only blanks. If the program was in the file file, then it could be compiled into an executable called deblank by using:

```
% lex file | rc -s deblank
```

For a complete description of lex, its flags, input file format, and output, see lextut(tutorial).
FILES

+LIB/lexskel - skeleton program to be filled out
+INCL/lskdef - definitions for skeleton program's actions

SEE ALSO

AUTHOR
Vern Paxson. Evolved from an original implementation by Jef Poskanzer, with the help of many ideas from Van Jacobson.

BUGS/DEFICIENCIES

Unix lex has action macros YYMORE, YYLESS, and YYREJECT. These have not been implemented.

Trailing context must have a fixed size (i.e., no use of '*', '|', '+', '{m,n}', '{m,}', or ' ' operators).

There must be no whitespace between the macro invocation of BEGIN and its argument. That is, "BEGIN( x )" is incorrect; "BEGIN(x)" is correct.
NAME
libdel - delete modules from a library

SYNOPSIS
libdel library modu1 [modu2-modu3 ...]

DESCRIPTION
Libdel deletes the specified module(s) from the given library. Specific modules and ranges of consecutive modules may be deleted. At least one module should be given. If a certain module is duplicated or is nonexistent, none of the given modules will be deleted.

FILES

SEE ALSO
libup(tool), lslib(tool)

AUTHOR
Marylou Orayani

BUGS/DEFICIENCIES
On the vax, if the library is not a legal VMS library, then libdel will hang because of a problem with spawning to the local dcl commands. A control-c or control-y will abort the program.
NAME
libind - create an index of library routines

SYNOPSIS
libind [file] [-] ...

DESCRIPTION
Libind prints a one line description for each library routine found in a library
source on to standard output. If no arguments are given library filenames
are read from standard input. If a dash ("-") is given as a file argument the
library is read from standard input. The output is in the following format:

    routine name - one line description       source-file

If the library is read from standard input "source-file" is the string "STDIN".

FILES
None

SEE ALSO
ratags(tool)

AUTHOR
Roland McGrath

BUGS/DEFICIENCIES
Currently, libind only works on ratfor and VAX-macro sources. The routine
name and one line description are entirely dependent on their being given in
the source file in the format:

    ### routine name - one line description

or for Macro routines:

    ;;; routine name - one line description
NAME
libup - tool to update libraries

SYNOPSIS
libup <object [-v] [-f] library

DESCRIPTION
Libup adds the program objects on standard in to the library specified. Any
programs which are already on library are replaced with the newer versions.
If library does not exist, it is created.

If the -v option is specified, a report specifying which library names were
added and replaced is written to standard output.

The -f option avoids the default time consuming library compression.

FILES
/tmp/<taskname>/lbnnn: temporary file to hold directives to LIB.

SEE ALSO
lslib(tool), libdel(tool), Max IV LIB manual.

AUTHOR
Vern Paxson

BUGS/DEFICIENCIES
Libup will attempt to update 'library' even if library is not a legal library (for
example, 'library' is really a source). The only clue that something is wrong
is a message from the LIB processor:

CHECKSUM ERROR

At this point you should type 'a' followed by a carriage return to abort the
LIB processor, and try again with a valid library name. Your file should not
have been altered.
NAME
lip - list program

SYNOPSIS
lip [-io] [-l <n>] [file] ...
   -i: list all include files
   -o: list all include files once
   -l <n>: print 'n' lines per page

DESCRIPTION
Lip lists the programs specified on standard output. If no programs are
specified or if only a dash is specified, lip reads from standard input.

If the -i flag is specified, lip lists every encountered include file, all the
include file's include files, and so on. If the -o flag is specified, lip only lists
each include file the first time it is encountered in the source, i.e., each
include file is only listed once. Lip begins each routine at the start of a new
listing page. If there is a formfeed in the source, lip starts a new page.

The number of each line is printed in the left margin. If the line is from an
include file, the depth of the include file is printed next to the line number.
The depth of the file is represented by a blank for the source file, 'a' for a file
included by that file, 'b' for a file included by a file included by the source
file, and so on.

Lip prints a header and a footer on each page, containing the routine name,
the file name, date, and page number. (It uses one page lookahead to deter-
mine the routine name.) The -i flag can be used to specify the page length.
The default page length is 66; the minimum page length is 6 (the size of the
header and trailer).

Hash marks or question marks are put in the margin of code that is condi-
tionally compiled out. A question mark is used if the code is in the range of
an ifdef, elseif, or enddef (see the rat4 manual entry). The user may selec-
tively list certain parts of the program by using the "#list" and "#noist"
directives. These must be the first non-blank tokens on the line, and may be
nested.

SEE ALSO
rat4(tool), pr(tool)

AUTHOR
Anne Herrmann
BUGS/DEFICIENCIES

Lip uses the end statement of a routine to determine the beginning of the next routine. The end statement is only recognized on a line by itself. Thus, if a routine does not have an end statement, or if there is (say) a comment on the same line, the following routine will appear on the same page.

Tokens are recognized regardless of embedded blanks. This means (in particular) that if a leading substring of a line happens to be a recognized token, strange behaviour may occur. For instance,

If the formfeed character appears in a line in the source, and is not the first character on the line, it is treated like any other character and copied to the output. This usually causes a page eject on the output device, resulting in a split page, and should therefore be avoided.
NAME
  ll - print line lengths

SYNOPSIS
  ll [-v] files...

DESCRIPTION
  ll prints the lengths of the shortest and longest lines in the named files. The
  -v flag causes it to print headings along with the numbers. The name "-" may
  be used to refer to the standard input. If no files are given, ll reads the stan­
  dard 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.

AUTHOR
  David Hanson and friends (U. of Arizona)
NAME
lock - check out files or directories

SYNOPSIS
lock [-c comment] {file | -d directory} ...

DESCRIPTION
Lock checks out the files (and possibly directories) specified. After successfully using lock, the user can modify the files checked out and be sure that nobody else will modify them.

-c The following argument is a short explanation of why the files following it are being checked out. The comment is used for all files following it until another comment supercedes it. Note that the comment must be one argument, so it is recommended that the user put the argument in quotes: for example

    lock -c "fixing bug in lock program" /usr/src/lock.r

-d The next file name in the argument list is a directory. A directory can't be locked unless the -d flag is specified.

FILES
/<directory>/ .lock - the file containing the list of files locked in that directory
/<directory>/ .owner - the owner of the directory /<directory> (and particularly the owner of /<directory>/ .lock)

SEE ALSO
unlock(tool), lslock(tool)

DIAGNOSTICS

file checked out already <etc>
The user tried to check out a file that has already been checked out. He will have to wait until the file is available again.

file in directory checked out <etc>
The user tried to check out a directory, but one of the files contained in that directory has been checked out. The lock tool might still let him check the directory out, but he must be aware that he can't modify the file described.

- <file>: check out successful
The lock program is able to check out the file specified. It will actually be checked out if there are no errors in checking out other files.
- `<file>`: checked out with exceptions noted
The lock tool is able to check out the directory specified, but is reminding the user that one or more files in that directory have been checked out already, and he can not modify these. The directory will actually be checked out if there are no errors in checking out other files.

- `<files>`: unable to check out
The lock tool can not check out the file specified, probably because it or a parent directory has been checked out already.

- `<file>`: warning - file doesn't exist
The user tried to lock the file specified, but it doesn't exist. The user may have misspelled the file name. Note that this message is only a warning, so the lock program might lock the file anyway if there are no other problems.

-- `<directory>`: can't lock directory unless `-d` flag given
*The lock tool can't lock a directory unless the user* has preceded the directory name by the `-d` flag.

- `<file>`: can't lock - invalid file name
The user tried to lock the file `<file>`, but there couldn't possibly be such a file, as the file name doesn't make sense.

- `/`: can't lock root directory
The user has tried to lock the root directory. The lock program doesn't know how to do this. He should lock the individual top level directories (for example `/usr`, `/sys`, `/etc`, `/mnt`) instead.

- `<file>`: can lock in only one project
The user has tried to lock files contained in two different top level directories in the same command. He should lock the files in separate commands.

?? can't `[re@]`open owner file
Either somebody else is using lock or unlock at the same time as you are, and you should try again later, or you have tried to lock a file in a top level directory that doesn't exist or doesn't have .lock and .owner files.

?? can't `[re@]`open lock file
Somebody is looking at the lock file. Try again later.

?? lock file garbaged
The lock file is not in the proper format, and the lock program is unable to do anything about it. Send mail to trouble as soon as possible.

?? fatal errors: no locks performed
There were errors in attempting to lock the files requested. *None of the
files that the user requested to lock has been locked.

_files specified have been locked_
Everything's ok, and all the files requested to be locked have been locked. Nothing is locked unless the user gets this message.

AUTHOR
Todd Hammond

BUGS/DEFCIENCIES
NAME
logout - exit shell

SYNOPSIS
logout

DESCRIPTION
Logout causes the user to exit the current shell.

An alternate way of exiting the shell is to type an end of file. An end-of-file is
~Z (control-z) on the vax, and '$$' followed by a newline on the modcomp.

SEE ALSO
login (tool), sh (tool)

BUGS/DEFICIENCIES
On the modcomp, an end-of-file given to the login shell will log the user off.
NAME
lookfor - look for a file using a standard search path

SYNOPSIS
lookfor [-smilv] file ...
   -s: search for +PATH/file
   -m: search for +MAN/file
   -l: search for +LIB/file
   -i: search for +INCL/file
   -l: search for +LIB/file
   -v: print verbose info about search

DESCRIPTION
Lookfor will search for each file named ‘file’ using search path environment variables. If the file is found, its full path name is written onto standard output. The search path lookfor uses is determined by the following flags:

   -s: simulate the sh, using the environment variable PATH.
   -m: simulates man, using the environment variable MAN.
   -l: simulate link and ld, using the environment variable LIB.
   -i: simulate includ, using the environment variable INCL.
   -v: print the search path and full path name for the file
       found, or print message if the file wasn’t found.

If more than one flag is specified, each corresponding search path is used to find ‘file’, and each find will result in a full path name being written to standard output, one path name per line. If no tool flags are specified, lookfor calls pathopen to determine if ‘file’ exists, thus allowing custom search paths to be specified.

SEE ALSO
splb(lib), files(info), paths(info)

AUTHOR
Theresa Breckon
NAME
lpr - queue a file to the printer

SYNOPSIS
lpr [-fnulv] [-c <n>] [file]...
     -f: no overstrike or underline
     -n: narrow printer queue
     -u: upstairs printer queue
     -l: label queue
     -v: verbose
     -c <n>: 'n' copies made

DESCRIPTION
lpr takes the named files (or standard input if none are specified or if just a
dash is specified) and queues copies of them to the printer. Unless the -f
option is specified, all overstriking and underlining in the documents which
have been achieved via backspaces are converted to the appropriate over­
strike lines to drive the printer. This is done by spawning to os which is
searched for using +PATH/os.

The -f flag omits the spawn to os, i.e. no overstriking or underlining is done.
This "fast" flag should be used with caution, as it processes the input file
directly. See the Bugs and Deficiencies section for further explanation.

The -n flag queues the files to the narrow paper queue (forms=1). The -u flag
queues them to the upstairs printer queue (forms=2). The -l flag will queue
the files to the label queue (forms=6). The default behavior of lpr is to queue
the files with forms=0.

The -v flag causes the arguments passed to os and the job number of the job
to be typed on ERRROUT at the successful queueing of the file.

The number of copies of the file to be queued may be specified using the -c
flag, i.e., -c3. The default is 1 copy.

SEE ALSO
os(tool)

AUTHOR
Joe Sventek

BUGS/DEFICIENCIES
WARNING: use the "-f" flag with caution. This "fast" flag causes the input file
to be processed directly. Caution should be used when the input file is a
scratch file, as in a pipe to lpr. The scratch file itself will be queued for print­
ing. If the scratch file has not started printing before lpr exits, it will be
deleted, and nothing will be printed. If there are no other files in the print
queue, there should be no problem.
WARNING: do not use the "-l" flag on the RTSG Vax. Since the RTSG Vax does not have such a printer with forms=6, files queued to the printer with this flag will never be printed. 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.

Note that when more than one file is printed at once, the files are merely concatenated and printed. Separate headers are not given, nor are form feeds inserted.
NAME
Irgen - translates LR grammar into parse tables.

SYNOPSIS
Irgen

DESCRIPTION
Irgen reads a modified LR grammar specification from standard input, and writes to standard output the generated parse tables. The output of yaclr is usually piped to this tool.

Input to Irgen is the same as the input to LR, with one extension. See the paper on LR (mentioned in "see also" section) for a full description on LR input. The extension to the LR input consists of an extra section. The sections are separated by a line containing only the characters "&P".

The first section contains token declarations. These declarations look exactly like the yacc token declarations, except literals must be in the LR form, (enclosed in brackets). The second section contains the LR grammar productions. These are as described in the LR paper.

The generated parse tables are output as fortran data arrays in common blocks. These tables are meant to be used in conjunction with the parse library, yyplb.

FILES
none

SEE ALSO
yaclr(tool), yyplb(lib), yc(tutorial), yacc(tool) "Yacc: Yet Another Compiler-Compiler" by S.C. Johnson "LR - Automatic Parser Generator and LR(1) Parser" by C. Wetherall and A. Shannon.

AUTHOR
Theresa Breckon

BUGS/DEFICIENCIES
NAME
ls - list file or directory information

SYNOPSIS
ls [-acdhlnprtu1] [name] ...

DESCRIPTION
ls lists information about each file name. When no name is given, the current working directory is listed. By default, the files are always listed in alphabetical order, regardless of how they are found in the directory. Wildcarding capabilities are allowed on some machines, see the bugs section below for a list of these machines. There are several options:

-a lists all files and directories. Default action is to not list any filenames beginning with a period ['.'].

-c lists names in columns instead of 1 per line. This option is the default if standard output is to a terminal. It is ignored if '-l' or '-1' is specified.

-d lists directory instead of listing the files it contains (used mainly in conjunction with '-l' to get information on a directory).

-h prints a header at the top of verbose listings.

-l list in long (verbose) form instead of giving only the file name (long form described below).

-n lists names alphanumerically.

-p lists full pathnames. Normally the listed names don't include the name of the resident directory.

-r reverses the sense of the sort.

-t sorts by time modified (oldest first). The -t option always produces a listing in the long form.

-u lists last usage date instead of last modification date (if -l or -t is specified).

-1 forces single column output to the terminal. The default is multi-column output to the terminal, single to a disk file.

The long form file information contains the date and time of last modification (default) or usage, the owner of the file, the size of the file (granules written for the Modcomp, blocks for the Vax), the machine-dependent file permissions, the name of the file, and one character giving the file type:

null - data file
/ - directory
SEE ALSO
sort(tool), istree(tool), wclb(lib)

AUTHOR
Vax version by Joe Sventek, Modcomp version by Van Jacobson; wild carding by Jane Colman.

DIAGNOSTICS
Vax: "Could not read file info for <file>". This error message results when the user specifies '-l' or '-t' but does not have read permission for additional file information on <file>.

BUGS/DEFICIENCIES
Each machine has these differences listed below:

Vax: The Vax has wildcard capabilities, see the manual for wclb for further explanation. The file permissions in the verbose listing are for the owner, the group, and the rest of the world. In each appropriate field, a single letter denotes the permission; 'r' for read, 'w' for write, 'e' for execute, and 'd' for delete. If a file has a null file name, ls will list it as such. If a file and a directory have the same name, ls will assume you mean the directory even if the filename has an extension. On the Vax, the -a option is the default and -u isn't available.

Modcomp: The Modcomp doesn't have wildcard capabilities. The permissions in the verbose listing refer to non-owners of the file. They are:

- R - read permitted
- W - write permitted
- E - exclusive read/write permitted
- P - protected read/write permitted
- D - file creation permitted
- C - file deletion permitted
- M - file maintenance permitted

R, W, E, and P permissions only apply to data files. D permission only applies to directories. C permission applies to any file. M is currently meaningless. A letter appears only if the associated permission is granted, otherwise a '-' appears (e.g., 'R-PE-----' represents a file with read, protected, and exclusive permissions). The ln VOL is used to access the file manager volume. Because the file manager doesn't allow a directory to be opened, this tool must be privileged. 'ls /' should list the names of the currently mounted volumes instead of listing the volume directory of the default (system) disk. The pseudo-directory '/dev' can't be listed (it doesn't really exist). On the Modcomp, the -n and -t options aren't available.
NAME
lslib - list the program names in a library

SYNOPSIS
lslib [-] [object ...] >list

DESCRIPTION
lslib lists the program names on object files (either assembler output or libraries) on standard output. If no arguments are given, or a '-' argument is given, standard input is read.

SEE ALSO
libup, libcat, libdel, librep, MAXIV LIB

AUTHOR
ern Paxson

BUGS/DEFICIENCIES
If one of the arguments is not really an object file, lslib simply doesn't write out any program names for that file.
NAME
lslock - list contents of lock file

SYNOPSIS
lslock [-l] [file/dir] ...
   -l: list lock comments

DESCRIPTION
Writes a list on standard output of the locked files and directories related
to the files and directories in the argument list. If the argument is a directory
then all locked files in that directory will be listed. For example, say that the
following files have been locked:
   /usr/man/lib/hislb.m
   /usr/man/tool/lslock.m
   /usr/src/lslock

Then "lslock /usr/man" would produce information about the files
   /usr/man/lib/hislb.m
   /usr/man/tool/lslock.m

and "lslock /usr | find lslock" would produce information about the files
   /usr/man/tool/lslock.m
   /usr/src/lslock

The output consists of three fields separated by tabs. The three fields contain
the locked file name, the login name of the lockee followed by the real
name in parentheses, and the date and time. For example,

   /usr/src/ls xero (Nancy Travis) 10Oct84 09:41:23.75

If the -l flag is given, the list includes any comments given by the users who
locked the files. These comment lines consist of zero or more lines of the form

   # <about a line of comments>

FILES
<directory>/lock - the file containing the list of files locked in that directory

SEE ALSO
lock(tool), unlock(tool)
DIAGNOSTICS

<file>: invalid file name
The file name specified is not a legal file name, so the lslock program is unable to look it up.

/: can't do list of files locked in root directory
The lslock tool does not know how to do an lslock on the root directory. Instead, say 'lslock /usr /sys /etc /mnt ...'.

?? can't open lock file
Somebody is looking at the lock file. Try again later.

?? lock file garbaged
The lock file is not in the proper format, and the lslock tool is unable to read it. Send mail to trouble as soon as possible.

AUTHOR
Todd Hammond

BUGS/DEFICIENCIES
NAME
lstape - list contents of backup tape

SYNOPSIS
lstape [-ial]

DESCRIPTION
lstape lists the contents of a backup tape which was written with the backup
and/or save tool on standard output. Each file is listed on a separate line.

The following flags are recognized:

- i List files in immediate mode. This is the default, since non-immediate
  mode is not currently supported.

- a List all versions of all files, instead of just the most recent. This is the
default in immediate mode.

- l Produce a long listing. The listing will look like

  <date> #<s>.<n> (<version>) <file>

The fields are separated by tab characters (although they look like they are
separated by spaces in this manual entry). <date> is the date and
time that <file> was written on the tape. <s> is the name of the saveset
on the tape which <file> is stored in, and <n> is the position of <file>
within the saveset <s>. <version> is the version number of the file on
the tape, which is one more than the number of files with the name
<file> preceeding the current <file> on the tape. <version> is '0' in
immediate mode. <file> is the full pathname of the file in tools format. If
the '-l' flag is not specified, just <file> is listed.

The user should not allocate or mount the tape drive. This is done by lstape.

FILES
/mt - the tape drive that the tape is read from

SEE ALSO
backup(tool), save(tool), restore(tool)

AUTHOR
odd Hammond

DIAGNOSTICS
BUGS/DEFICIENCIES
Non-immediate mode should be implemented.

Dec version numbers should not be listed. Instead they should be included in the tape version number. For example, if the tape contained 'file;1' then 'file;2' then 'file;1' then 'file;3', the first file should be considered version 1 of 'file' on the tape, the second file should be considered version 2 of 'file' on the tape, the third file should be considered version 3 of 'file' on the tape, and the fourth file should be considered version 4 of 'file' on the tape.

The lstape tool is different in several ways from the lstape tool on the modcomp. The modcomp lstape tool allows the user to specify (with wildcards) which files to list. Also, on the modcomp, non-immediate mode is the default.

The name of the device where the files on the tape came from is not stored on the tape. Thus, lstape must guess what the device where the files originally came from is. Lstape assumes that a file on the tape came from the disk that the current working directory is on, unless the top level directory of the file is something like /usr, /etc, or /misc. Thus, if your current working directory is on /mnt, lstree will usually assume that all the files on the tape are from the /mnt drive, and if your current working directory is on /dra0, lstree will usually assume all the files are from /dra0.

The date given for an entry is actually the date and time when the saveset started to be written, not the date and time that the entry was written.
NAME
Istree - list file system directory sub-tree

SYNOPSIS
Istree [-adhlpu] [name] ...

DESCRIPTION
Istree lists portions of the file system directory tree starting with the supplied file name(s) (if no names are given, the list starts with the current working directory). The list consists of the names of all files below the given file in the directory hierarchy.

Available options are:

- `a` lists all files and directories. Default action is to not list any filenames beginning with a period `['.'].`, and to not list a subtree whose full pathname contains a `'/.'`.

- `d` includes directories in the list (normally the list consists of only non-directory files).

- `h` prints a header at the top of verbose listings.

- `l` lists in 'long' (verbose) form instead of just giving the file name. Long form includes the file's protections, owner name, size, and last modification (or usage) date in addition to its name.

- `p` lists complete pathnames. Normally the listed names don't include the name of the resident directory.

- `u` lists last usage date instead of last modification date (if `-l` specified).

The long form file information is shown as:

- The date and time of latest modification (default) or usage.
- The owner name.
- The size of the file (granules written for Modcomp, blocks for Vax).
- The machine-dependent permissions.
- The name of the file.
- One character giving the file type:
  
  null - data file
  / - directory
  - - partition (Modcomp only)
  ? - interleaved (Modcomp only)
SEE ALSO
ls(tool)

AUTHOR
Modcomp version by Van Jacobson, Vax version by Vern Paxson.

DIAGNOSTICS
"Could not read file info for <file>". This error message is generated whenever 'l' has been specified and a given file does not have read permission for the user; lstree is unable to access any file information other than the file's name.

BUGS/DEFICIENCIES
Each machine has these differences:

Vax:

Owner, group, and world protections are listed. In each appropriate field, 'r' denotes read permission, 'w' denotes write permission, 'e' specifies execute permission, and 'd' indicates delete permission.

Extensions in argument directory names are ignored. The command

    lstree incl.sp

is equivalent to

    lstree incl

On the Vax, the -a option is the default and -u isn't available.

Modcomp:

The permissions listed with an -l flag are for non-owners:

R - read permitted
W - write permitted
E - exclusive read/write permitted
P - protected read/write permitted
D - file creation permitted
C - file deletion permitted
M - file maintenance permitted

R, W, E, and P permissions only apply to data files. D permission only applies to directories. C permission applies to any file. M is currently meaningless. A letter appears only if the associated permission is granted, otherwise a '-' appears (e.g., 'R-PE----' represents a file with read, protected, and exclusive permissions).
The lfn VOL is used to access the file manager volume.

Because the file manager doesn't allow a directory to be opened, this tool must be privileged.
NAME
m - format a programmer's manual entry

SYNOPSIS
m [-i] file ...
   -i: format implementor section

DESCRIPTION
M uses roff, rtsgmac and manmac to format standard programmer's manual
entry(s) (see the documents mentioned below for a description of the RTSG
standard programmer's manual and its implementation on various machines
and projects).

At least one filename must be given (i.e., m can't take its input from a pipe).
The formatted document is written to standard output.

The -i flag signifies that an implementors manual is requested and m will for-
mat the implementors section as well. Otherwise the implementors section of
the manual is ignored.

FILES
/usr/lib/rtsgmac - rtsg standard document macros
/usr/lib/manmac - rtsg standard manual macros

SEE ALSO
manual(info),f(tool),roff(tool),manmac(lib) st.2.2.1 - Introduction to RTSG
Unix Document Tools st.2.2.4 - Compatible Document Prep Commands (Sum-
mary) st.2.2.5 - Macros for Standard Document Preparation

BUGS/DEFICIENCIES
There is currently a limit of ten files to format.
NAME
macro - process macro definitions

SYNOPSIS
macro [-d] [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 a language or 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)

"Name" can consist of letters, digits, and underlines. 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 replaced by a null string.

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

\[
\text{define(d,[define($1,$2)])}
\]

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:

\[
\text{define}(x,y) \\
\text{define}(x,z)
\]

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

\[
\text{define}(x,y) \\
\text{define}([x],z)
\]

Besides "define", macro has 11 other built-in functions. Note that all of these function names are reserved words. E.g., the word "ascii" will always invoke the ascii function unless enclosed in "[...]", or escaped with an "@".

\section*{arith}

This performs simple arithmetic functions:

\[
\text{arith}((\text{operand1,op,operand2})
\]

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

\[
\text{define(had,[arith($1,+,\$2)])} \\
\text{add}(5,3)
\]

would produce the result '8'. Because this function is so awkward to use, a similar but much more convenient function called "math" was installed.

\section*{ascii}

This takes one argument and returns a numeric string equal to the USASCII code of the first character of the argument. Thus,

"ascii(A)" evaluates to "65", 
"ascii(a)" evaluates to "97", 
"ascii(0)" evaluates to "48", and 
"ascii( )" evaluates to "32".
chr  This is the reverse of ascii - it takes a number and returns the corresponding character. Examples:

"chr(32)" evaluates to ",
"chr(incr(ascii(A)))" evaluates to "B", and
"chr(10)" evaluates to a NEWLINE.

error  This will write its argument to ERROUT and abort the program; it is exactly equivalent to the Software Tools routine of the same name.

evaluate  This performs parameter substitution on its first argument using 2nd and later parameters as arguments. E.g.,

\texttt{evaluate(-$1- -$2-,dog,cat)}

would return "-dog- -cat-" as replacement text. Evaluate is the basic primitive for constructing loops or iteration. For example, a macro to evaluate its first argument with each of the following arguments could be:

\texttt{define(foreach,[ifelse($2",[evaluate([$1],$2)@
foreach($1,$3,$4,$5,$6,$7,$8,$9)])])}

If this were invoked as:

\texttt{foreach([draw-rail($1)],K4,RL07,KE14)}

the result would be

\texttt{drawrail(K4)}
\texttt{drawrail(RL07)}
\texttt{drawrail(KE14)}

ifelse  This is a conditional test. The input

\texttt{ifelse(a,b,c,d)}

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

\texttt{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.

incr  This converts its argument to a number, adds one to it, and returns that as its replacement text (as a numeric character string). "incr" can be used for tasks like
define(MAXCARD,80)
define(MAXLINE,incr(MAXCARD))

which makes two parameters with values 80 and 81.

length  This returns as a numeric string the number of characters in its argument.

math    This is an easy-to-use replacement for arith. You give it an expression in standard infix notation and it evaluates it and returns the result. It has the operators + (addition), - (subtraction), * (multiplication), / (division), and ^ (exponentiation). It handles parentheses, negative numbers, and real numbers. There is no operator precedence - all evaluation is left to right, except as indicated by parentheses. Examples:

"math(60-(23/2))" evaluates to "48.5",
"math(2^10)"    evaluates to "1024", and
"math(2^-10)"   evaluates to "0.976563E-03".

repeat This takes its first argument and repeats it the number of times specified by its second argument. Examples:

"repeat(Foo,5)" evaluates to "FooFooFooFooFoo",
"repeat(),10)" evaluates to "",
"repeat(junk,0)" evaluates to the null string.

substr This takes 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)" evaluates to "b",
"substr(abc,2)" evaluates to "bc", and
"substr(abc,4)" evaluates to a null string.

Two other features worth noting:

- The -d flag puts macro into a debugging mode where the names of all macros are printed out on ERROUT as they get evaluated. The names are indented to reflect how deep the call is nested.

- ATSIGN (@) can be used to 'escape' almost anything. If a line ends with an ATSIGN, neither the ATSIGN nor the NEWLINE will be seen. This is very useful in complex macro definitions. ATSIGNs before a reserved
word or character remove the special meaning of the word or character. E.g., "@define" keeps the word "define" from being recognized. Like square brackets, one ATSIGN is stripped each time macro reads the input. Thus, "@@" becomes "@". Unfortunately, an argument substitution ("$n") can't be escaped with either brackets or an ATSIGN. The only way to prevent an argument substitution is to enclose the "$" in brackets (e.g., "[$]3").

As a final example, here is how you had to compute the length of a string before the built-in version of length was installed:

\[
\text{define}(\text{len}, [\text{ifelse}(\$1, 0, [\text{incr}(\text{len}(\text{substr}(\$1, 2)))])] )
\]

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.

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 300.

- call stack overflow - The maximum level of nesting of macro calls has been exceeded. Currently this is 130.

- EOF in string - An end-of-file was read while looking for the closing bracket of a bracketed string.

- evaluation stack overflow - The total number of characters for name, definition, and arguments has been exceeded. Currently this is 1500.

- unexpected EOF - An end-of-file was read while searching for the closing paren of a macro call.

- filename: can't open - For some reason, the file specified could not be opened - perhaps it does not exist.

- too many characters pushed back - Macros work by "pushing back" characters on the input stream. This error means that some sequence of macros attempted to push back more 1500 characters. (It takes some combination of macros to get this error since one macro will overflow the evaluation stack before it gets a chance to overflow the push-back buffer).
AUTHOR
From "Software Tools" by Kernighan and Plauger, with minor modifications by Debbie Scherrer. Many functions, new symbol table routines by Jef Poskanzer & Van Jacobson.

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.

There should be a way to truncate a real number. All the built-in functions that read integers - arith, chr, incr, repeat, substr - ALMOST do the right thing. They quit reading their integer when they get to a decimal point, so they do correct truncation for all cases except E-notation.
NAME
Mail - utility for sending mail to local users

SYNOPSIS
mail [-f message file] [-s subject] address [address]

DESCRIPTION
Mail permits a user to send mail to the addresses provided in the argument list. It is especially useful for sending mail from a program, since the body of the message will be read from standard input. If the standard input is not redirected, 'stmail' is spawned to permit the user to compose the mail.

If the '-f' option is specified, the body of the message will be read from the specified file and sent to the specified user(s). If the '-s' option is specified, the subject of the command argument is placed in the "Subject:" field of the message. If the subject string consists of more than one word, enclose the text in quotation marks ('').

SEE ALSO
users - a program to list users on current host
postmn - a program which notifies user of existence of mail
msg - the utility for sending and receiving mail.
stmail - sending mail utility

AUTHOR
Whei-Ling Chang. Originally written by Joe Sventek.

BUGS/DEFICIENCIES
NAME
man - display the manual entries for the tools specified

SYNOPSIS
man [-n] name ...

DESCRIPTION
Man is a tool which locates and prints (onto standard output) the manual
entry described by "name". If standard output is a terminal, the output is
paused after every page displayed and the user is prompted for instructions.

If a -n argument is specified (where n is a numeric string), the value of n is
assumed to be the number of lines on the terminal screen. The current
default is 39 lines.

The search path specified in the environment variable MAN is used by man to
find the desired manual entries.

SEE ALSO
help(tool), usage(tool), UNIX man

DIAGNOSTICS
"man: can't find x" means either the manual entry for x couldn't be found or
it was found but couldn't be read.

AUTHOR
The original man was written by Van Jacobson. Search paths were incor­
porated by Bob Upshaw.

BUGS/DEFICIENCIES
Man uses the +MAN search path environment variable to find a manual entry.
The shell uses a different variable, +SHELL. Therefore, the manual entry
found by man may not correspond to the tool by the same name found by the
shell. The user should keep his/her environment variables +PATH and +MAN
compatible.
NAME
mcol - multicolumn formatting

SYNOPSIS

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" charac-
ters are truncated. (The default column width is 60.)

-sn Same as -w.

-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

AUTHOR
Original by David Hanson and friends (U. of Arizona), with modifications by Debbie Scherrer (LBL).
NAME
  memo - format memo

SYNOPSIS
  memo [-d] file ...

DESCRIPTION
  Memo prepares a memo based on the text in the files specified.

  If the `-d' flag is used, the memo is a draft, and will have a line saying that it
  is a draft printed in the memo.

  The following macros are recognized in the text:

  @.loc <room>, <building>, <extension>
  The room, building, and extension of the author of the document. If
  this macro is not used, the room, building, and extension are not
  printed.

  @.date <date>
  The date of the document. If this macro is not used, or if this
  macro is used but the date is not specified, the date is not printed.

  @.category <category>
  The category or document number of the memo. For example, the
  argument to the 'category' macro might be 'Software Tools', or
  'RT83-999'. If this macro is not used, the category is not printed.

  @.to <group>
  The people to whom the memo is addressed.

  @.from <who>
  The person (or people) who wrote the memo.

  @.subject <subject>
  The subject of the memo.

  @.longsubject
  The following text will be used for the subject lines. If an argument
  is given to this macro, it is used as the first line of the subject, and
  used on the header line. Otherwise, no subject is put on the header
  line. If you don't want the subject lines to fill, you must type the
  first line of the subject, then type @.nf, and then type the other
  lines of the subject.

  @.endsubject
  The end of the subject lines.
These macros must be called in the order specified. The user is free to use the standard 'f' and 'roff' commands, such as @.pp and @.lp, @.bd and @.br.

SEE ALSO
   f(tool), m(tool), roff(tool)

AUTHOR
   Todd Hammond

BUGS/DEFICIENCIES
   Memo can't process more than 10 files at a time.

   To use commas in arguments, you must escape them with '@' symbols.

   There is a maximum of 4 people on the to and from lines. To specify more, you must escape the commas.
NAME
mk - make files

SYNOPSIS
mk [-d] files ...

DESCRIPTION
mk takes each of its arguments and tries to create a file under that name. If a file already exists, mk prints a message to that effect and goes on to the next file, if any, unless the -d flag is specified. In that case, mk does not check to see if the named file exists; any existing file is deleted, and re-created.

FILES
None.

SEE ALSO
UNIX mk, mkdir
mkdir(tool)

DIAGNOSTICS
Yes, if -d is not specified and a file exists; if no arguments are specified; if a file can't be created.

AUTHOR
Anne Herrmann
NAME

mkdecsym - generate include file for dec symbols

SYNOPSIS

mkdecsym name

DESCRIPTION

Makes a rat4 include file incl/name which defines the symbol definitions in
the macro include file $name. However, the user should use the %loc conven­
tion for symbol definitions whenever possible.

AUTHOR

Todd Hammond

BUGS/DEFICIENCIES

Occasionally, macro is unable to include some existing symbol file. This
causes mkdecsym not to be able to generate an include file for that file.
NAME
mkdir - make directory

SYNOPSIS
mkdir directory ...

DESCRIPTION
Mkdir creates the specified directory or directories. It is optional for the
specified name to end with a slash.

Example:

mkdir junk

This will create the directory "junk/" in the current directory.

mkdir /mnt/harold/bin

Assuming that the directory /mnt/harold already exists, this will create the
directory "bin/" in the directory /mnt/harold.

SEE ALSO
rmkdir

DIAGNOSTICS

"mkdir: Failed to create directory - name/"

This message may occur because the directory name is invalid or because it
is a privilege violation to create the directory.

AUTHOR
Craig Leres

BUGS/DEFICIENCIES
NAME
mktags - make a tags file

SYNOPSIS
mktags [-e] [-v] [file] ...

DESCRIPTION
Mktags generates output from ratfor program and Vax-11 Macro sources that can used as an editor tags file. If no file arguments are specified, input is read from STDIN and the string "STDIN" is used as the file specification in the generated output.

A tags file gives the locations of programs, subroutines, and functions in a file or group of files. Each line of the tags file contains the routine name, the file in which it is defined, and an address specification for the object definition (in this case, a search string). The three fields are separated by white space, for example:

dospwn /usr/src/ftljb /%integer function dospwn/

Normally, the output of mktags should be sorted with sort(tool), for example:

% mktags /usr/src/sh | sort > tags

Mktags accepts the following flags:

-e By default, mktags constructs address specifications using Tools regular expressions. The "-e" flag causes mktags to use regular expression compatible with the Unix editor ex.

-v This flag causes mktags to produce an index of the form expected by Unix vgrind. This listing contains the function name, file name, and page number (assuming 64 line pages).

FILES

SEE ALSO
sort(tool)

AUTHOR
Craig Leres

BUGS/DEFICIENCIES
Mktags doesn’t know about ifdef.
NAME
Mretry - re-submit mail saved by composition programs

SYNOPSIS
mretry [file] ...

DESCRIPTION
Mretry attempts to re-submit messages which were saved by 'stmail' and
'locmail' when the mail delivery system did not respond to the delivery
request. (This action should only be necessary if the mail system was not
restarted after a reboot of the system, or if there is some dire problem with
the mail system.) If no filename is specified, 'mretry' looks for the file
'dead.ltr' in the user's home directory, which is the place where the composi-
tion utilities place dead letters. If the re-submission is successful, the file is
deleted.

The composition utilities and 'mretry' expect the file to have a very special
format, so the 'file' argument should only be specified if the user has
renamed ~/dead.ltr.

FILES
~/dead.ltr

SEE ALSO
stmail - mail composition utility

DIAGNOSTICS

AUTHOR
Joe Sventek. Modified by Wei-Ling Chang.

BUGS/DEFICIENCIES
NAME
Msg - utility for sending and receiving mail.

SYNOPSIS
msg [-fn] [-p[n]] [file]

DESCRIPTION
Msg provides a friendly environment for sending and receiving mail. It offers
the user simple editing capabilities to ease the composition of outgoing mes-
sages, as well as providing the ability to read, write and modify files which
have the message file format.

The msg system accepts incoming messages for you from other people and
collects them in a file, called mymail. When you login, the system notifies
you if there are any messages waiting in your mymail file. When you read
your mail, it reads your mymail file and separates that file into the individual
messages that have been sent to you. You can then read, reply to, delete,
forward to or save these messages. Mbox is the file where the messages from
mymail are saved by default.

Finally, msg is able to send and receive messages across networks such as
the DECDNET and ARPANET.

Msg is entered via the following command line:

   msg [-p[n]] [filename]

If no filename is specified, msg defaults to read the file mymail in the home
directory.

The -f switch is used to change the number of fill characters written to the
terminal whenever a carriage-return, line-feed pair is written. The -p switch
can be used to change the page size (number of lines per screenful) for 'msg'
to use when displaying long output. The default is 22 lines.

The valid msg commands are listed as follow:

   a[nswer]   message
   b[ackup]   to previous message and type it
   c[urrent]  message number and file
   d[el]      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[gnore]   header field(s)
   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 a file and mark them deleted
   n[ext]     message is typed
FILES

mymail - messages sent using the 'stmail' utility reside here.
mbox - default file for saving messages from mymail.

SEE ALSO

mail - the utility for sending mail to local users
msplit - the utility for salvaging message files

AUTHOR

Whei-Ling Chang. Originally written by Joe Sventek.

BUGS/DEFICIENCIES
NAME
Mspli - utility for salvaging message files

SYNOPSIS
msplit <file [-v] [root]

DESCRIPTION
Mspli reads the standard input file, which presumably has more messages on it than msg can handle, and splits it up into files which contain a fixed number of messages (currently the maximum number of messages in one file is 500). The files created are named rootaa, rootab, ..., where root defaults to "tmsg" if none is specified on the command line. If the -v (verbose) option is selected, then the name of each file is displayed on ERROUT as it is created. You should immediately use msg on these temporaries to place the messages in appropriate files. In particular, the input file to msplit should be overwritten, since it is too large already.

FILES
none

SEE ALSO
msg - the utility for sending and receiving mail

AUTHOR
Joe Sventek

BUGS/DEFICIENCIES
NAME

delta - make a 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 preceding version are computed and the TCS file will be able to reproduce either version (or earlier versions) by means of the get(tool) 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(tool), get(tool), Unix delta(tool).

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.

"Revision file is empty". Perhaps an incorrect filename was given.

"History file is empty". The first formal version should be entered by means of the admin(tool) command.

"Line stack overflow." "Run stack overflow." The "diff" algorithm has used up all of its internal storage. This indicates that the revision is very different from the most recent version in the history file (currently to get one of these error messages requires 3000 lines to be different between the two files).

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

"Temp file error: (filename)"). The temporary file created during processing disappeared unexpectedly.
AUTHOR
Original code by Wil Baden; converted from MORTRAN by Dave Murray. Modifications and conversion to BTL-SCCS style by Neil Groundwater at ADI. Diff algorithm replaced by Vern Paxson.

BUGS/DEFICIENCIES
File permissions are NOT manipulated 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(tool), lpr(tool)

AUTHOR
Original from Kernighan & Plauger's 'Software Tools', with modifications by Dennis Hall and Debbie Scherrer.
NAME
diff - list differences between two files.

SYNOPSIS
diff [-cw] [-m lines] old_file [new_file]

DESCRIPTION
 Diff will compare two files and summarize their differences. The output of
 Diff is similar to the Ed directives to convert "old_file" to "new_file":

 i,j d m  Lines i thru j in old_file deleted
 from new_file. The deleted lines
 went after line m of new_file.

 i a m,n  Lines m thru n were added to new_file.
 The lines were added after line i
 of old_file.

 i,j c m,n Lines i thru j of old_file were
 changed to lines m thru n of
 new_file.

 All of the numbers to the left of the command are line numbers in old_file
 and numbers to the right are line numbers in new_file. Following each direc­
tive line is a list of the associated lines. The old_file lines are all preceded by
 the characters "<-" and the new_file lines by the characters "->".

 If new_file is omitted, the new file is assumed to be standard input. The file
 name "-" also denotes standard input.

 The flag "-c" causes case (i.e., upper/lower) to be ignored when comparing
 lines. E.g., "ABC" would be considered equivalent to "abc".

 The flag "-w" causes white space (blanks and tabs) to be ignored when com­
 paring lines. E.g., "a b c" would be considered equivalent to "abc".

 The flag "-m lines" sets the number of lines that must be matched to
 "resync" the two files (the default is three). If this number is too small, diff
 will be fooled by short sequences of common code (e.g., <blank
 line>/return/end/<blank line>) will resync diff and cause it to report more
 changes than actually exist.

FILES
 None.
SEE ALSO

comm(tool), ed(tool) Unix diff

DIAGNOSTICS

appears if the stack which keeps track of "runs" of matching lines overflowed. This usually means that the two files are completely different. However, it may occur if diff improperly resyncs - try something like "-m 6" to see if this the case.

AUTHOR

Van Jacobson

BUGS/DEFICIENCIES

Unix diff has a "-e" flag which causes the actual edit directives to convert file1 to file2 to be output.

NAME
dspc - display all characters in a file

SYNOPSIS
dspc [files ...]

DESCRIPTION
Dspc copies the named files to the standard output, replacing control characters by "\~c", where c is the ascii character used to enter the control character when depressing the "control" key on most terminals. More precisely, c is the ascii character whose value is 100 (octal) plus the value of the control character.

Control characters are those ascii characters with values between 0 and 31, inclusive. Newline characters are displayed as dollar signs ($) and delete characters as \?.

If no filenames are given, or the name '-' is specified, dspc will read from the standard input.

SEE ALSO
   cat(tool) crt(tool) the 'l' command in ed(tool)

AUTHOR
   David Hanson and friends (U. of Arizona)
NAME
   echo - echo command line arguments

SYNOPSIS
   echo [arg ...]

DESCRIPTION
   Echo writes its arguments in order as a line on the standard output file. Each argument is separated by a single BLANK character. It is useful for producing messages and diagnostics in command files.

   If there are no arguments, nothing is output.

SEE ALSO
   The Unix command 'echo' in the Unix manual

AUTHOR
   Debbie Scherrer
NAME
ed - text editor

SYNOPSIS
ed [-] [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.

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-searching 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 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).

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

8. 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 current line. End of File or '.' terminates the append mode noting that the current line is now the last line of the appended text.

(.)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 preceeding (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. End of File or '.' terminates the change mode noting that the current line is now the last line of the changed text.

(....)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.

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. A global continues even if one or more of its command fails on a given line. Multiple commands may be executed by placing each on a preceding line and terminated each command except the last with an atsign '@'.

(.)i

<text>

The insert command inserts the given text BEFORE the current line. End of File or '.' terminates the insert mode noting that the current line is now the last line of the inserted text. This command differs from the a command only in the placement of text.

(...)<address>

The copy command makes a copy of the text in the range specified, and puts it after <address>.

(...)<address>

The list command lists the lines specified unambiguously; that is, all embedded non-printing characters are translated into ~char, where char is a corresponding printing character. A control a becomes a ^A, an ascii value of 0 a ^@, and a delete character a ^?, for example. The end of the line is marked with a dollar sign ($). The current line becomes the last of the listed lines.

(...)<address>

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

(...)<address>

The number command prints the given lines preceded by their line numbers. The last line printed becomes the current line.

(...)<address>

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.

The quit command causes ed to exit. Note however that if no write has been made since the last change was made to the file, ed will
warn you and won't let you quit. To quit no matter what, type q!

(.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. If no delimiters occur after the s command (i.e. 'n,ms'), then the last substitute-from and substitute-to patterns that were used are assumed. Global substitution is done if it was on the previous substitution. '.' 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 '@'.)

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

(1.$o

The overlay command allows the user to replace characters within a line(s). After issuing the overlay command, the current line is echoed to the terminal. The user then types in a line containing a space in each character position not to be changed. Each character typed by the user, which is beyond the last character in the line being modified is appended to the end of that line(s). The modified line is echoed back to the terminal. If more than one line is overlayed on a given command, only the last line is echoed. The overlay command gives the user the ability to directly manipulate any desired characters on a line. Whereas, this in not always the case with the substitute command, which changes only the first
occurrence of a pattern or all occurrences of the pattern within a line.

After entering the overlay mode, three (3) special characters '"', ' ', and '~' are used for replacing characters with blanks, deleting characters, and inserting characters, respectively.

The following paragraph is used to illustrate the features of the overlay command.

This paragraph illustrates the ability of the overlay command to manipulate characters within a line(s). Mistakes are made for illustration purposes.

a. overlay the current line.

:o<cr>
of the overlay command to manipulate
a
:o<cr>
of the overlay command to manipulate

b. overlay a line and append to it.

:o<cr>
characters within a line(s). Mistakes
s i s are<cr>
characters within a line(s). Mistakes are

(Notice that the delimiter has no effect if the range of the line is exceeded).

c. Overlay several lines.

(line, line)<delimiter>

### delim - sample routine for overlay
#
integer function delim(char)
integer char

    if ( char != ' ' )
        delim = 1
    else
        delim = 2
    return
end
:o<cr>
end <current line>
end <current line>

end 02c<cr> 02c
<resulting lines of code>:

```c
### delim - sample routine for overlay

integer function delim(char)  
external
integer char

if (char != ' ')  
delim = 1
else

delim = 2
return
end
```

For examples 'd' through 'f' it is assumed the user is already in overlay mode.

d. Replace a character with a blank.

```c
line of text
& cr>
```


e. Delete a character from a line

```c
character to be deleted.  
< cr>
character to be deleted.
```

f. Insert character into a line

```c
Line in which to insert a character.
- o < cr>
```

(Characters are inserted before the position pointed to by '.'). Insertion is terminated by either of the following characters: ' ', '@n', EOF, or '.').

(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.

(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.

&

Repeats the last command typed. Note that no address may be given to the & command; it uses the address of the last command as well.

# 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.

SUMMARY OF SPECIAL CHARACTERS

The following are special characters used by the editor:

<table>
<thead>
<tr>
<th>Character</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Matches any character (except newline)</td>
</tr>
<tr>
<td>%</td>
<td>Indicates beginning of line</td>
</tr>
<tr>
<td>$</td>
<td>Indicates end of line or end of file</td>
</tr>
<tr>
<td>[...]</td>
<td>Character class (any one of these characters)</td>
</tr>
<tr>
<td>[!...]</td>
<td>Negated character class (any character expect these characters)</td>
</tr>
<tr>
<td>*</td>
<td>Closure (zero or more occurrences of previous pattern)</td>
</tr>
<tr>
<td>@</td>
<td>Escaped character (e.g. @%, @[, @*)</td>
</tr>
<tr>
<td>&amp;</td>
<td>Ditto, i.e. whatever was matched</td>
</tr>
</tbody>
</table>
Range of characters between c1 and c2

Specifies the newline character at the end of a line.

Specifies a tab character

FILES
A temporary file is used to hold the text being edited.

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

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

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

BUGS/DEFICIENCIES
At the present time the editor is still in a somewhat experimental version. It has not been overlayed yet, nor have any real attempts been made to improve its efficiency. For these reasons, the field length needed for execution is high and the files being edited are limited to 1000(VMS-5000) lines. The line limit applies to all lines read in and subsequently changed. For instance, a 600 line file with 401 changed lines would exceed ed's buffer space. 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.

Because of the nature of the editor's IO, it is usually impossible to properly print an altered line whenever its newline character has been deleted or a new one inserted. Line numbering and counts may also become erratic, although the file itself is actually correct. This problem can be alleviated by writing (w command) and re-editing (e command) the file after tampering with any newline characters.
NAME
edt - execute visual editor edt

SYNOPSIS
edt [-inrv] file

DESCRIPTION
Edt executes the DCL visual editor "edt" to edit specified file. Edt is normally run from a vt100 or vt52 terminal (or suitable emulator).

The following flags are recognized:

-i The file is opened for input only. Edt does not attempt to write into the file. This flag is useful, for example, to look at a file which you have write permission to, or a file you don't want to modify. Otherwise, the editor will note that you don't have modification access to that file, and will exit.

-r Recover: this option can be used if the user was using edt while the system crashed, if the user aborted edt with a control-c or control-y, or if edt was otherwise abnormally terminated. There is, however, no protection for accidental mistakes that the user makes, such as quitting edt, when he intened to write out his file, and so forth. To use the recover option, type in the same command line to edt that you used before, but add the flag -r.

-n Nocommand: this option can be used when the user does not wish to execute the edt initialization file ~/edtini.edt.

-v Verbose: this option causes the edt tool to display the command line it uses to execute DCL edt.

If no file name is given, edt edits the file 'edt' in the current working directory.

The edt initialization file ~/edtini.edt is executed by edt before it executes unless the -n flag is specified.

FILES
~/edtini.edt - edt initialization file.

SEE ALSO
See 'Vax-11 EDT Editor Reference Manual' for information on how to use edt.

Edt also has interactively available documentation. To see this documentation in non-visual mode, get into edt, and say 'help'. In visual mode, help is also available. On a vt100, the user may type the sequence <PF1><7 on keypad>HELP<enter> to get the same help information as he would get in non-visual mode by typing 'help'. On the vt100, the user may also type
<PF2> to get help with the commands available for visual editing.

Type "dcl 'help edit/edt'" for information on how to run edt from dcl.

DIAGNOSTICS

AUTHOR
    Todd Hammond

BUGS/DEFICIENCIES
    Typing control-C will cause the editor to abort. Typing control-Y can get you into worse trouble.
NAME
entab - convert spaces to tabs and spaces

SYNOPSIS
entab [ <t1> ... ] [ +<n> ] [-a] [file...]

DESCRIPTION
Entab replaces strings of blanks with equivalent tabs (control-i) and blanks.
Tab stops are indicated by <t1>... 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. By default only leading
blanks are converted to maximal strings of tabs. If the -a option is given,
tabs are inserted whenever they would compress the resultant file by replac­
ing two or more characters. If no files are specified, the standard input is
read. An isolated minus sign also indicates the standard input.

SEE ALSO
detab(tool), lpr(tool)

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

BUGS/DEFICIENCIES
None to date.
NAME
ext - terminate sh command file

SYNOPSIS
goto{tool}
exit

DESCRIPTION
Exit performs a seek to the end of its standard input file. Thus, if it is invoked inside a file of commands, upon return from exit the shell will discover an end-of-file and terminate.

SEE ALSO
sh{tool}, goto{tool}, if{tool} The UNIX command exit

AUTHOR
Bob Upshaw
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(tool)

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

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

BUGS/DEFICIENCIES
NAME
  f - format an RTSG standard document

SYNOPSIS
  f file ...

DESCRIPTION
  F uses roff and rtsgmac to format a document in the RTSG standard form
  (see the documents mentioned below for a description of the RTSG standard
  document form and its implementation on various machines).

  At least one filename must be given (i.e., f can't take its input from a pipe).
  The formatted document is written to standard output.

FILES
  /usr/lib/rtsgmac - rtsg standard document macros

SEE ALSO
  m(tool),roff(tool) st.2.1.1 - The RTSG Standard Document Format st.2.2.1 -
  Introduction to RTSG Unix Document Tools st.2.2.4 - Compatible Document
  Prep Commands (Summary) st.2.2.5 - Macros for Standard Document
  Preparation

BUGS/DEFICIENCIES
  There is currently a limit of ten files to format.
NAME
   fb - search blocks of lines for text patterns

SYNOPSIS
   fb [-acx] [-in] [-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

-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 con-
        sidered the pattern which ends a block (e.g. -send).

-li  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(tool), ed(tool)
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"

AUTHOR
Debbie Scherrer (Lawrence Berkeley Laboratory)

BUGS/DEFICIENCIES
An expression may not start with a minus sign (-). Regular expressions can not span line boundaries.
NAME
fc - fortran compile and link

SYNOPSIS
fc [-bedlmotv] [-a arg] [-f arg] [-g arg] [-s arg]
[program] [libraries...]

DESCRIPTION
Fc is a tool to compile and optionally link fortran programs. The first argument it encounters that is not a flag is assumed to be the name of the fortran source. Any subsequent arguments are assumed to be names of libraries. Fc uses the following flags.

-a the following argument is given to the assembler
-b just make a library
-c just make an object
-d make a debuggable output (binary, object, or library)
-f the following argument is given to the fortran compiler
-g the following argument is given to the linker or librarian
-l make a listing file
-m make a loadmap
-o keep the object file
-s the following simple name is used for the name of the output files (binary, etc.)
-t do not link with the standard software tools library.
-v verbose mode. Fc will print all spawns to errout as they occur.

The b and c flags may not be used together. If neither -b nor -c is specified, an executable file is written onto the bin subdirectory. The -b and -o flags are also mutually exclusive.

The output files are written to specific directories with specific extensions. The "name" below is the argument following the -s flag. If a name is given with -s, it must be a simple name without extensions. If no name is given with -s, the name defaults to the last simple name of the source file, without extensions. If no source file name was given, the file name "fcout" is used.

The output files are:
Fc recognizes the file name '-' to be the standard input for the source program, but not for the libraries. Also, if no program or libraries are given, fc will read from standard input.

Libraries specified will be expanded using the library search path, unless the library is specified with a full pathname.

FILES
+LIB/library - the standard tools library

SEE ALSO
rc(tool), ld(tool), rat4(tool), ratp1(tool), ratp2(tool), comlb(lib)

AUTHOR
Marshall Spight

BUGS/DEFICIENCIES
Since the vax doesn't need to invoke the assembler to do fc's job, the information given with the -a flag is ignored.
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 com­
mas (e.g. 5-10,18,72+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). If neither a fieldlist nor a delimiting character is supplied, '-t'
with a tab is assumed.

Once fields have been described they can be arranged on output by the 'out­
putformat' 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. (The argu­
ment $0 refers to the whole line.) These $n symbols are placed in the output
format wherever that field should appear, surrounded by whatever charac­
ters desired. For example, an outputformat of:

"$2 somewords $1"

would produce an output line such as:

   field2 somewords field1

The normal escapes are allowed in the output format (e.g., '@n' for newline,
'@t' for tab, '@f' for formfeed). If a '"S' is to appear in the output, it must be
escaped with a '"'. If no input files are specified, or if the filename '-' is
found, field will read from the standard input.
NAME
mtr - read files from tape

SYNOPSIS

DESCRIPTION
Mtr reads named files from tape in raw Unix format in either ascii or ebcusic, or it reads blocked records in either character set. These tapes are writeable by mtw. If no file names are specified on the command line, the program reads the file names from standard input.

The possible switches are:

-v print each file name on standard output as it is being read from tape

-s Normally, trailing blanks are stripped from each line. If -s is specified, they are left intact.

-fu At most, u files are read. If two empty files (i.e. three end-of-files in a row) are read before u files are read, mtr will halt. If -f is alone is specified, or -f0, 1 is used. Default is a large number (10000).

-ni causes mtr to interperate the character with the ASCII code of i as an end-of-line. (E.g. -n0 causes mtr to interperate a null byte as a newline.) If -n is specified, -n0 is used. Default is NEWLINE.

-e expect ebcodic codes on the tape. The default is ascii.

-bn:j read blocked records from the tape. 'j' specifies the number of characters in each line image. (Default is 80) 'n' specifies the number of line images packed into 1 record on the tape (default is 1).

-m If mtr can't write on some file, it should make its own name. Normally, mtr aborts if it can't write a file. The name of the file will be of the form 'mtrfile.nnn', where 'nnn' is the sequence number of the file on the tape.

-d Overwrite file if it already exists. Otherwise, mtr asks the user whether it should replace the file or not. (Like the '-d' flag in 'cp'.)

If there are not enough file names for all the files on the tape (or the maximum number of files to read specified by the -f flag), mtr makes up names of the form 'mtrfile.nnn', where 'nnn' is the sequence number of the file on the tape.

The tape is not rewound before mtr starts reading.

To use on the VAX:
1. hang tape on drive
2. $ allocate mta0:
3. $ mount/foreign/density=800 mta0:
   (you can use other densities also)
4. Get into the sh and say:
   % mtr [your arguments]

SEE ALSO
mtw(tool), wmodt(tool), rmodt(tool)

AUTHOR
Originally written by Joe Sventek. Modified by Bob Upshaw.

BUGS/DEFICIENCIES
NAME
mtw - write file to tape

SYNOPSIS
mtw [-v] [-rn] [-e] [-bn:j] [-f] [file ... ]

DESCRIPTION
mtw writes named files to tape in raw Unix format in either ascii or ebcdic, or it writes blocked records in either character set. These tapes are then readable by mtr. Each file is written to the tape and terminated by a tape­mark. The end of information is encoded as 3 successive tapemarks after the last file. If no file names are specified in the command line, the program goes into prompt mode and reads file names from the standard input. This mode of operation is terminated by an EOF on standard input.

The switches allowed are:

-v print each file name on standard output as it is being written to tape

-rn each record on the tape is n bytes long. The default value is 1920. The specified value is forced to lie in the interval [20,15360] and is forced to be even.

-e output ebcdic codes on the tape. The default is to output ascii.

-bn:j write blocked records to the tape. Each line image will be j characters long (default is 80), and n line images will be packed into 1 record on the tape (default is 24). The line length is forced up to an even value if it is odd. The product n*j is forced to lie in the same interval as for the -rn switch, and if these bounds are exceeded, the blocking factor n is adjusted to bring the product into the interval.

-f write filled blocks. Blocks are filled with the character 255(8). By default, if there aren't enough records to fill a block, the block just ends.

To use on the VAX:

1. hang tape on drive
2. $ allocate mta0:
3. $ mount/foreign/dens=800 mta0:
4. Get into the sh and say:
   % mtw [your arguments]
SEE ALSO
mtw(tool), wrnodt(tool), rmodt(tool)

AUTHOR
Originally written by Joe Sventek. Modified (to work with a Modcomp, mostly) by Bob Upshaw.

BUGS/DEFICIENCIES
The tape is not re-wound before or after mtw executes.

If tapes are written in ebcidic and are filled, mtw attempts to write convert the fill character to ebcidic. Mtw will either abort or give you garbage, depending on how lucky you are.

If fill characters are not written and the block is too small (around 20 characters), mtw will not be able to write the blocks and will abort. Thus, unblocked (i.e. raw unix) output doesn't work unless the -f flag is specified.
NAME

mv - move or rename files

SYNOPSIS

mv [-dv] file1 file2
mv [-dv] file directory

DESCRIPTION

Mv moves (changes the name of) file1 to file2.

If file2 already exists, mv will query the user with

mv: file2 already exists - replace?

and wait for a yes/no response before continuing (this query is always directed to the user's terminal, even if mv is being used in a script). The question can be defeated by using the -d (destroy) flag in which case mv will copy without checking whether the destination file exists.

In the second form, one or more files are moved to the directory with their original file-names.

The -v (verbose) flag forces extra information to be printed about the moving of each file.

SEE ALSO

cp(tool), cptree(tool)

AUTHOR

Craig Leres

BUGS/DEFICIENCIES

If file1 and file2 lie on different file systems, mv copies the file and deletes the original. This can be dangerous if the copy fails in a non-fatal manner since the source may be deleted.

Normally, mv will refuse to copy a file onto itself. However, sometimes it is possible to lose a file if you specify a version number. For example:

mv -d junk..1 junk..0

will clobber junk..1 if it is the only version.
NAME

note - leave a note for a reminder

SYNOPSIS

note [-d display_condition] [-e erase_condition]

DESCRIPTION

Note allows a user to leave personal messages (called "notices") to be displayed and erased at specific times. Note is used to add notices to the users "notice" file, and the corresponding tool "notify" is used to display and/or erase notices. The supplied "display_condition" is used to specify the times a notice is to be displayed onto standard output, and the "erase_condition" specifies the time to erase the notice from the notice file. If a display_condition is not specified, "always" is assumed. Likewise, if an erase_condition is not given, "never" is assumed.

The display_condition and erase_condition, if given, are boolean expressions which will evaluate to TRUE or FALSE. ('Always' always evaluates to TRUE, and 'never' always evaluates to FALSE.) When the tool "notify" passes thru the notice file, it evaluates the two expressions. If the display_condition evaluates to TRUE, the corresponding notice is written to standard output. If the erase_condition evaluates to TRUE, the notice is removed from the notice file. The display condition is always acted upon before the erase condition.

The boolean expressions are constructed using variables concerned with the current time and date. The following syntax is allowed:

expression ::= secondary { "&" secondary }
secondary ::= primary { "|" primary }
primary ::= 
  ( ('expression') )
  relation
  'always'
  'never'
relation ::= arithprim relop arithprim
relop ::= '==' | '!=' | '<' | '>' | '<=' | '>='
arithprim ::= integer_constant # any integer
             symbolic_constant
             time_variable
symbolic_constant ::= sunday | monday | tuesday | wednesday
                     | thursday | friday | saturday
                     | january | february | march | april
                     | may | june | july | august | september
                     | october | november | december
time_variable ::= month # current month (1-12 or jan-dec)
                 | day  # the day of the month (1-31)
                 | year # the current year (e.g. 1982)
                 | dow  # the day of the week (1-7)
                 | hour # the hour of the day (0-23)
                 | minute # the minute of the hour (0-59)
(Note: Only the first four letters of a keyword are significant - all others are ignored. All keywords are case-insensitive.)

Notices

After note has verified the validity of the boolean expressions, it reads lines from standard input (up to the EOF.) The lines are assumed to be notice text, except in the case of:

(1) A line which begins with a shell escape character (!) causes a shell to be spawned to execute the rest of the line. If the rest of the line is empty, the user is placed into an interactive shell. After the spawned shell terminates, note continues to read notice text.

(2) A line which begins with an escape character (@) is passed to the notice file directly. (Thus, one can have a notice line begin with a '!' which can be useful. See the tool notify).

Examples:

% note

(will cause the input notice text to always be displayed and never erased.)

% note -d 'dow = Thurs'
Don't forget - Meeting Today at 4pm.
<EOF>

(will remind the user every Thursday of an important meeting.)

% note -d 'day>=20 & month>=novem' -e 'day>=25 & mond=decemb'
What are you doing at work? You should either be skiing or Christmas shopping.
<EOF>

FILES

~/.notice.txt (on the Vax) - contains the users notice file.
SEE ALSO
   notify(tool), mail(tool), msg(tool)

DIAGNOSTICS

AUTHOR
   The ratfor version was written by Bob Upshaw. However, many ideas were taken from a Pascal program named 'memo' at Georgia Tech.

BUGS/DEFICIENCIES
   There is no convenient way to erase notices once they have been put into the notice file (unless a -e argument was specified, and then only when it's time.) However, the notice file can be modified with the editor; a brief description of the format of the notice file is given in the manual for notify.
NAME
notify - notify the user of any outstanding notes

SYNOPSIS
notify

DESCRIPTION
When notify is executed, it passes thru the users notice file (prepared by the
note tool), looking for notices waiting to be displayed and/or erased. Any
notices which are displayed cause all lines of that notice file to be written to
standard output with one exception: Any line which begins with a shell escape
code character (!) will cause that line to be executed by a spawned shell. When
notify completes, any notices which should be erased will have been removed
from the users notice file.

An invocation of notify is usually placed in ones login startup file.

To facilitate the user modification of the notice file (via the editor), a brief
description of the file will be given:

The first line of the file contains the date the file was last written - usually the
last time the file was passed over by notify. THIS LINE MUST BE PRESENT AS
IT IS ALWAYS REMOVED BY NOTIFY, AND REPLACED WITH A CURRENT DATE.
Beginning with the second line of the file, all notices begin with four lines:

(1) -e followed by the erase condition as specified as an argument to note.
(2) The code for the erase condition as generated by note.
(3) #d followed by the display condition as specified as an argument to
note.
(4) The code for the display condition as generated by note.

The rest of the lines up to a line beginning with a '-' or an end-of-file are the
lines of the notice. Each line begins with a '.' (which is stripped off by notify
before being processed.)

FILES
~/notice.txt (on the Vax) - contains the users notice file.

SEE ALSO
note(tool), msg(tool)
DIAGNOSTICS

AUTHOR

Bob Upshaw, with many of the ideas taken from 'memo' of Georgia Tech.

BUGS/DEFICIENCIES
NAME
os - overstrike - convert backspaces into multiple lines

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(tool)

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

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

BUGS/DEFICIENCIES
NAME
pack - pack words into columns

SYNOPSIS
pack

DESCRIPTION
Pack takes the words (groups of characters separated by blanks or tabs) found on the standard input and outputs them to standard output in five columns, 16 spaces wide, ordered from left to right. The characters used to achieve the separation of columns are TAB characters, so that those terminals which support hardware tabs can be driven efficiently.

FILES

SEE ALSO

DIAGNOSTICS

AUTHOR
Joe Sventek

BUGS/DEFICIENCIES
NAME
  pc - pascal compile and link

SYNOPSIS
  pc [-bcdlmotv] [-g arg] [-p arg] [-s arg]
      [program] [libraries...]

DESCRIPTION
  Pc is a tool to compile and optionally link pascal programs. Pc uses the
  "includ" tools to expand ratfor-style include directives in the source file. The
  first argument it encounters that is not a flag is assumed to be the name of
  the pascal source. Any subsequent arguments are assumed to be names of
  libraries. Pc uses the following flags.

  -b just make a library
  -c just make an object
  -d make a debuggable output (binary, object, or library)
  -g the following argument is given to the linker or librarian
  -l make a listing file
  -m make a loadmap
  -o keep the object file
  -p the following argument is given to the pascal compiler
  -s the following simple name is used for the name of the output files
      (binary, etc.)
  -t do not link with the standard software tools library.
  -v verbose mode. Pc will print all spawns to errout as they occur.

  If neither -b nor -c is specified, an executable file is written onto the bin sub-
  directory.

  The output files are written to specific directories with specific extensions.
  The "name" below is the argument following the -s flag. If a name is given with
  -s, it must be a simple name without extensions. If no name is given with -s,
  the name defaults to the last simple name of the source file, without exten-
  sions. If no source file name was given, the file name "pcout" is used.

  The output files are:

  library (-b):     lib/name
  object (-c):      lib/name
list (-l): src/name.lis
loadmap (-m): src/name.map
binary : bin/name

Pc recognizes the file name '-' to be the standard input for the source program, but not for the libraries. Also, if no program or libraries are given, pc will read from standard input.

Libraries specified will be expanded using the library search path, unless the library is specified with a full pathname.

FILES
    +LIB/library - the standard tools library

SEE ALSO
    fc(tool), ld(tool), fc(tool), includ(tool), couib(lib)

AUTHOR
    Marshall Spight

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.

AUTHOR
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
Postmn - report the presence of mail

SYNOPSIS
postmn [-h [file]]

DESCRIPTION
Postmn looks to see if the requestor has received any mail. If so, it reports
that fact upon the standard output file with the comment
You have Software Tools mail

If the '-h' flag is specified, the message headers for the messages are
displayed on standard output, instead of the comment above. If the optional
trailing file name is specified, the headers for the messages in that file are
displayed.

FILES

SEE ALSO
stmail - utility for sending mail
msg - utility for sending and receiving mail
resolve - utility for querying the mail database about users
users - utility for listing valid mail users

AUTHOR
Joe Sventek

BUGS/DEFICIENCIES
NAME
pq - print disk quota

SYNOPSIS
pq [-h] [-a disk] [username] ...

DESCRIPTION
Pq prints out disk quota information. By default, it gives the quota for the current user.

The "-h" flag causes a header to be displayed.

The "-a" flag is used to display information about all users on the specified disk. Disk names are usually specified as "/dra0" or "dra0:"

Output is separated by tabs and is of the form:

used  total  username
100   1000  guest

A list of the top disk users on a might be generated with:

% pq -a /dra0: | sort -rn

AUTHOR
Craig Leres

BUGS/DEFICIENCIES
Users may have quotas on more than one disk; pq only reports about the disk the home directory is on.
NAME
pr - paginate files for printing

SYNOPSIS
pr [-h hdr] [-l pagelen] [-ln] [-po offset] [-t] [file ...]

DESCRIPTION
Pr produces a paginated, titled listing of one or more files. The output is titled with the name of the file (or a user specified header if the "-h" option is used), the date, time and page number. If there are no file arguments, pr prints its standard input & thus is usable in a pipe.

Options are:

-h treat the next argument as a header to be used instead of the file name.

-l treat the next argument as the page length (the default page length is 66 lines).

-ln each line is printed with a line number (the first line in each file is numbered '1').

-po treat the next argument as a 'page offset': All lines printed will have 'page offset' leading blanks. The default page offset is 0.

-t do not print the 4 line page header or trailer.

Each new file begins on a new page. Any formfeeds in the input also result in a new page.

FILES

SEE ALSO
cat(tool) Unix pr(tool)

AUTHOR
Van Jacobson

BUGS/DEFICIENCIES
Unix pr has several options related to multi-column output which would be nice to have:

-n produce n-column output.

-m print all files simultaneously, one in each column.

-w set the page width to next arg (used to compute space between columns).
-s separate columns by character following the 's'.

-2-
NAME
printenv - list the user's environment

SYNOPSIS
printenv [name] ...

DESCRIPTION
Printenv is a tool which lists the current set of environment variables and their respective values to standard output. Environment variable names are case-insensitive. If no environment variable names are specified on the command line, all environment variables are listed in the form "name=value". Otherwise, only the specified variables' values will be listed. For example,

```% printenv PATH TERM
~/.bin ~/.proc /usr/bin /misc/bin
vt52
```

```% printenv
PATH=~/.bin ~/.proc /usr/bin /misc/bin
INCL=~/.incl /usr/incl
TERM=vt52
USER=tab
```

AUTHOR
heresa Breckon

SEE ALSO
setenv(tool), resetenv(tool), unsetenv(tool), envlb(lib), sh(tool)

DIAGNOSTICS

"printenv: Can't list environment variables" - An error occurred while trying to list all of the environment variables. You should consult your software manager.
NAME
  prlabl - format labels for printing

SYNOPSIS
  prlabl [-width] < filename

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 7 lines or less. 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.

AUTHOR
  Joe Sventek
NAME
profile - profiler for VAX/VMS programs

SYNOPSIS

DESCRIPTION
Profile is a tool which samples time spent in another program.
The flags are as follows:
-1 Report about hits in p1 space.
-2 Report about hits in p2 space.
-3 Report about hits in p3 space.
-a Read a local ".stb" file, if it exists.
-g<F> Change the time between peeks to "<F>" (a real number). The default is 0.5 seconds. This value may be too large for programs with short execution times, however running profile with a granularity less than 0.5 seconds will cause the system to thrash and will distort reported statistics. The accepted method for testing small programs is to give them a larger input to process.
-h Produce a short help message that explains flags for profile.
-l<S> Make buckets for each line of code in the routine named "<S>" (a string).
-p<X> Profile the process with process id "<X>" (a hexadecimal number).
-r Read sys$system:rmsdef.stb for routines. This flag implies "-1".
-s Read sys$system:sys.stb for routines. This flag implies "-1" and "-2".
-t<F> Change the routine display threshold to "<F>" (a real number). The default is 0.1 percent, which means that routines that use less than 0.1 percent of the total cpu time are not displayed.
-w Display bucket list.

Either a program or a process id must be specified. If a pid is specified, the file spec of the program is found with sys$getjpi(). In either case, the program is read and buckets are created based on the routines specified in the traceback records.
The output `profile` generates includes percent time spent in processor states and percent time spent in routines.

Here's an example usage of `profile`:

```bash
% profile -lm main ratp2 @<junk.in @>junk.out >junk.pro
```

Notice that '@'s must be used to escape redirection arguments for the program to test.

**SEE ALSO**

`sh(tool)`

**DIAGNOSTICS**

There are many, but none are listed here.

**AUTHOR**

Craig Leres

**BUGS/DEFICIENCIES**
NAME
prompt - prompt a user for input

SYNOPSIS
prompt [-] [string] ...

DESCRIPTION
Prompt is used to get strings from the users terminal. Each argument is output to the terminal and a line is read from the terminal and echoed to STDOUT. If the argument "-" is encountered, or if no arguments are specified, the prompt strings are taken from STDIN.

SEE ALSO
prompt(lib)

AUTHOR
Craig Leres
NAME
ps - list process status information

SYNOPSIS
ps [-axl] [-t<S>] [-u<S>] [<X>]

DESCRIPTION
Ps displays information about processes on the system. The information
displayed includes username, process id, terminal name, process state, cpu
time, base and current priorities, and simple image name. By default, ps
lists processes associated with the current terminal.

Ps accepts the following flags:
-a List all processes associated with a terminal.
-x List all processes.
-l Display the full pathname of the image.
-t<S> List all processes associated with the terminal "<S>" (a string).
-u<S> List all processes associated owned by the user "<S>" (a string).
<X> List the information about the process "<X>" (a hexadecimal number).

The cpu time is displayed as seconds and hundredths, minutes and seconds,
hours, or days, depending on the magnitude.

Sample output:

% ps
leres 018c0037 ttf1 lef 4/7 1:55 sh
leres 01140032 cur 4/4 0.23 ps
% ps -l
leres 018c0037 ttf1 lef 4/6 1:55 /usr/bin/sh
leres 01150032 cur 4/4 0.26 /usr/bin/ps

SEE ALSO
who(tool)

DIAGNOSTICS

"ps: Not installed with WORLD."

This error message occurs when ps is not installed properly. In order for
non-privileged users to run ps, it must be installed with WORLD privilege.
When this error occurs, complain to the system manager.

"ps: sys$getjpi(), <vms error text>"
March 19, 1985

This indicates that something unforeseen has occurred. If you get this message, please report the status code to the system manager.

AUTHOR
Craig Leres
NAME
  pwd - print the user's current working directory

SYNOPSIS
  pwd

DESCRIPTION
  Pwd prints the name of the user's current working directory onto standard
  output.

FILES

SEE ALSO
  cd(3), mkdir(3)

AUTHOR
  ob Upshaw

DIAGNOSTICS

  Can't get working directory
    Pwd is unable to find the name of the working directory.

BUGS/DEFICIENCIES
  The internal working directory is assumed to be the same as the external
  one.
NAME
rat4 - translate ratfor into fortran

SYNOPSIS
rat4 [-knv] [-s arg] [program]

DESCRIPTION
Rat4 is a tool to translate ratfor programs into fortran. The first argument it
encounters that is not a flag is assumed to be the name of the ratfor source.
Any subsequent arguments produce an error. Rat4 uses the following flags.

-k keep the fortran file
-n do not include the standard definitions file
-s the following simple name is used for the name of the fortran output file,
   if any.
-v verbose mode. Rat4 will print all spawns to errout as they occur.

The output is written to the standard output unless the -k flag is given, in
which case the file is written onto src/name.f. The "name" is the argument
following the -s flag. If a name is given with -s, it must be a simple name
without extensions. If no name is given with -s, the name defaults to the last
simple name of the source file, without extensions. If no source file name was
given, the file name "rat4out" is used.

Rat4 recognizes the file name '-' to be the standard input for the source pro­
gram. Also, if no program name is given, rat4 will read from standard input.

FILES
 INCL/ratdef - the standard tools definitions

SEE ALSO
fc(tool), ld(tool), rat4(tool), ratp2(tool), comlb(lib)
ratpl(tool) - for a description of ratfor syntax

AUTHOR
Marshall Spight

BUGS/DEFICIENCIES
NAME
ratfix - fix a source file for the new ratfor

SYNOPSIS
ratfix < oldsrc > newsrc

DESCRIPTION
Ratfix is a filter that converts old-style ratfor source into the new STUG standard ratfor. It handles quoted string conversion, LETA -> 'a', inserts DRIVER and DRETURN calls, and more! Ratfix was specifically written for RTSG ratfor but should not have any difficulties with files from other sources.

SEE ALSO
ratp1(tool), ratp2(tool), rat4(tool), rc(tool)

DIAGNOSTICS
"warning - initr4 called on line %d."

"Conditional compilation code detected on line %d, requires manual clean-up." (%equal, %or, %not, %and)

"%if detected on line %d - almost certainly will not work anymore."

"%elseif detected on line %d - almost certainly will not work anymore."

AUTHOR
Vern Paxson and Marshall Spight

BUGS/DEFICIENCIES
Handling of conditional compilation is not perfect. The old style conditionals checked to see if the macro given was defined as YES, and the new style just check to see if the macro is defined at all. Code that makes significant use of conditional compilation may require significant changes by a human.
NAME
ratp1 - pass one of ratfor to fortran translator

SYNOPSIS
ratp1 [-n] [-] [file] ...
   -n: don't include standard definitions
   -: read from standard input

DESCRIPTION
Ratp1 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.

The second pass of the translation is performed by the ratp2 tool. This pass reorders the fortran code to be ANSI 66 compliant.

Syntax:
Ratfor has the following syntax:

```plaintext
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 argu-
ment 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 interpretation 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:

- \texttt{arith(x,op,y)} performs the "integer" arithmetic specified by \texttt{op (++,\,-,\,*,/,**)} on the two numeric operands and returns the result as its replacement.
- \texttt{incr(x)} converts the string \texttt{x} to a number, adds one to it, and returns the value as its replacement (as a character string).
- \texttt{ifelse(a,b,c,d)} compares \texttt{a} and \texttt{b} as character strings; if they are the same, \texttt{c} is pushed back onto the input, else \texttt{d} is pushed back.
- \texttt{substr(s,m,n)} produces the substring of \texttt{s} which starts at position \texttt{m} (with origin one), of length \texttt{n}. If \texttt{n} is omitted or too big, the rest of the string is used, while if \texttt{m} is out of range the result is a null string.
- \texttt{lentok(str)} pushes the length of the argument (\# of characters) onto the input as a character string.
- \texttt{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

```c
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

```c
string name "character string" or
string name(size) "character string"
```

declare 'name' to be a character array long enough to accommodate 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 utilizing 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 them with the tokens %(...%). The surrounding
%[(0)] tokens will be removed and the character sequence will be output
without change. Macro invocations, long names, etc. appearing in the charac-
ter 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 gen-
erated 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 thereof) 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 'undefined' 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.

A generalized definition file (e.g. 'ratdef') is automatically opened and read.

SEE ALSO
ratp2(tool), rat4(tool), rc(tool), includ(tool), macro(tool)
Kernighan and Plauger's "Software Tools"
Kernighan's "RATFOR - A Preprocessor for a Rational Fortran"
The Unix command rc in the Unix Manual

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 give 'return' an argument from a subroutine.
   A return statement in a subroutine was given a value. Subroutines cannot return values.
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.
- conditional processing still active at EOF
   A sufficient number of "enddef" directives have not been encoun-
   tered before detecting EOF on the input file.
* Conditions 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.
- 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
- illegal break
   Break did not occur inside a valid "while", "for", or "repeat" loop
- illegal case or default
   A "case" or "default" statement was detected which was not in the
   scope of a "switch" statement.
- illegal case syntax
   The case label was not of the correct form. It may consist of
   comma-separated constants or ranges of constants.
- illegal else
   Else clause probably did not follow an "if" clause
* illegal enddef encountered
   An "enddef" directive was encountered while conditional prepro-
   cessor was inactive.
- illegal next
   "Next" did not occur inside a valid "for", "while", or "repeat" loop
- illegal range in case label
A case label specifying a range of values (of the form m-n) was detected in which m > n.

- illegal right brace
  A right brace was found without a matching left brace

- in entdef: no room for new definition
  There is insufficient memory for macro definitions, etc. Increase the MEMSIZE definition in the preprocessor.

- 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.

- 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.

- invalid for clause
  The "for" clause did not contain a valid init, condition, and/or increment section

- 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" or "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.

- 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.

- 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 Rat-
   for) 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 process-
   ing 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 process-
   ing has been enabled.

* non-alphanumeric name
   Definitions may contain only alphanumeric characters and under-
   scores.

* 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 han-
   dle. 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 pushback buffer is set by BUFSIZE in the preprocessor symbols file.

- unbalanced parentheses
  Unbalanced parentheses detected in a Fortran (as opposed to Ratfor) statement

- 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.

- 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.

**AUTHOR**

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 Ratp1 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 - pass two of ratfor preprocessor

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
ratp1(tool), rat4(tool)

AUTHOR
Phil Scherrer
NAME
rc - ratfor compile and link

SYNOPSIS
rc [-bcdklmnotv] [-a arg] [-f arg] [-g arg] [-s arg] 
[program] [libraries...]

DESCRIPTION
Rc is a tool to compile and optionally link ratfor programs. The first argument it encounters that is not a flag is assumed to be the name of the ratfor source. Any subsequent arguments are assumed to be names of libraries. Rc uses the following flags.

-a the following argument is given to the assembler

-b just make a library

-c just make an object

-d make a debuggable output (binary, object, or library)

-f the following argument is given to the fortran compiler

-g the following argument is given to the linker or librarian

-k keep the fortran file

-l make a listing file

-m make a loadmap

-n do not include the standard definitions file

-o keep the object file

-s the following simple name is used for the name of the output files (binary, etc.)

-t do not link with the standard software tools library.

-v verbose mode. Rc will print all spawns to errout as they occur.

The b and c flags may not be used together. If neither -b nor -c is specified, an executable file is written onto the bin subdirectory. The -b and -o flags are also mutually exclusive.

The output files are written to specific directories with specific extensions. The "name" below is the argument following the -s flag. If a name is given with -s, it must be a simple name without extensions. If no name is given with -s, the name defaults to the last simple name of the source file, without
extensions. If no source file name was given, the file name "rcout" is used.

The output files are:

- library (-b): lib/name
- object (-c): lib/name
- fortran (-k): src/name.f
- list (-l): src/name.lis
- loadmap (-m): src/name.map
- binary: bin/name

Rc recognizes the file name '-' to be the standard input for the source program, but not for the libraries. Also, if no program or libraries are given, rc will read from standard input.

Libraries specified will be expanded using the library search path, unless the library is specified with a full pathname.

FILES

+LIB/library - the standard tools library
+INCL/ratdef - the standard tools definitions

SEE ALSO

fc(tool), ld(tool), rat4(tool), ratp1(tool), ratp2(tool), comlb(lib)

AUTHOR

Marshall Spight

BUGS/DEFICIENCIES

Since the vax doesn't need to invoke the assembler to do rc's job, the information given with the -a flag is ignored.
NAME
repeat - repeatedly execute a shell command

SYNOPSIS
repeat ntimes command

DESCRIPTION
Repeat will execute the shell command "command" "ntimes" or until it fails. For example,

    repeat 10 "echo hi there"

would print "hi there" 10 times on your terminal.

FILES
None.

SEE ALSO
sh(1) Unix repeat(1)

AUTHOR
Van Jacobson
NAME
resetenv - re-set your environment

SYNOPSIS
resetenv

DESCRIPTION
resetenv is a built-in shell command which resets your shell environment.
resetenv removes all current environment information and then executes
the user's "login" file. The "login" file is normally where the user keeps all of
the default "setenv" commands.

AUTHOR
heresa Breckon

SEE ALSO
printenv(tool), unsetenv(tool), setenv(tool), envib(lib), sh(tool)

DIAGNOSTICS
"resetenv: Couldn't reset user's environment." - Error occurred while trying
to reset user's environment and process the "login" file. The software
manager should be notified.
NAME
Resolve - resolve mail system user names

SYNOPSIS
resolve expression [expression ...]

DESCRIPTION
Resolve searches the mail database for lines matching the text pattern "expression". (Valid text patterns are the same as for find.) Resolve will display on the standard output all lines which match any one of the given expressions.

FILES

SEE ALSO
find - search a file for text patterns
mail - mail facility for sending mail to local users
users - list users to whom mail may be sent

DIAGNOSTICS
none

AUTHOR
Joe Sventek. Modified by Wei-Ling Chang.

BUGS/DEFICIENCIES
NAME
   restore - restore specified files/directories

SYNOPSIS
   (see below)

DESCRIPTION
   Restore is a tool that hasn't been implemented yet.

However, all is not lost. With a few simple hints you can actually restore your
files using the VMS backup utility. (Note that the backup tool uses the VMS
backup utility to save your files.) What follows is a general description of the
layout of a typical backup tape, a list of commonly useful commands, and a
suggested approach to recovering files from a backup magtape.

A backup tape consists of one or more backup save sets. Each save set con­tains some files. Every time you run the backup script, a new save set is
written after the last save set on the tape.

Here is a list of operations that are useful in the recovery of files from a
backup tape:

Mounting the backup tape. Before you can use a backup tape, you must phy­sically mount it on the tape drive and type the following:

   $ mount /foreign mta0:

Dismounting the backup tape. Before you can remove a backup tape from
the tape drive, you must type the following:

   $ dismount mta0:

Rewind a tape:

   $ set magtape /rewind mta0:

List the contents of the next save set on the tape to the terminal. This com­mand is useful for finding which save set contains the file you are looking for.

   $ backup /norewind /list=tt: mta0:

Advance one save set. This command is used to skip over the current save
set. No output is generated.

   $ backup /norewind /list=nla0: mta0:
Restore files from tape to the current directory. This command is used to recover the specified files from the next save set on the tape. Files are specified using VMS file specifications and wildcarding is allowed. The files are restored to the current working directory.

```bash
$ backup /norewind /select=([leres.src]ed.*.*) mta0: []
```

Typically, a file recovery session goes as follows. The backup tape is mounted on the tape drive. Save sets are listed on the terminal one at a time until the desired files are located. Next the tape is rewound to its beginning and the tape is advanced to the save set that is before the save set that contains the desired files. The files are then restored to a scratch directory. Finally, the tape is rewound and dismounted.

SEE ALSO
backup(tool), the VMS Utility reference manual
Online information is available thru the VMS help command

AUTHOR
Craig Leres
NAME
  resume - resume a suspended process

SYNOPSIS
  resume processid [processid ...]

DESCRIPTION
  Resume resumes a suspended process which has been suspended by the util­
  ity suspnd. The processid's are returned by the shell when a background
  process is spawned.

FILES
  None

SEE ALSO
  suspnd(tool) - suspend a running process sh(tool) - shell (command line
  interpreter)

DIAGNOSTICS
  If the process cannot be resumed, an error message will be displayed on the
  error output.

AUTHOR
  Joe Sventek (VAX)

BUGS/DEFICIENCIES
NAME
  rev - reverse lines

SYNOPSIS
  rev [files ...]

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.

AUTHOR
  David Hanson and friends (U. of Arizona)
NAME
revl - reverses the order of lines in a file.

SYNOPSIS
revl [file ...]

DESCRIPTION
Revl reads lines from standard input (or named files) and writes those lines to standard output in the reverse order. In other words, the first line of standard input becomes the last line of standard output, etc.

If no files are given, or the filename '-' is specified, revl reads from standard input.

FILES
none

SEE ALSO
cat (tool), crt (tool), rev (tool)

DIAGNOSTICS
A message is printed if a file cannot be opened; further processing of the tool is terminated giving no output. If a file is too large to be reversed, an error message is printed, and the tool is aborted.

AUTHOR
Jerry Cockrell

BUGS/DEFICIENCIES
Because of the use of dynamic storage, the size of the file to be reversed is limited by the machine.
NAME
   rm - remove files

SYNOPSIS
   rm [-aqv] [name...]

DESCRIPTION
   Rm removes the files specified. The file names are taken from the command
   line arguments. If no arguments are given, or the filename "-" is given, Rm
   will read the filenames from the standard input, one a time. The options
   work as follows:

   -a ask mode. Rm will ask the user about each file. A response of 'y' means
   to go ahead and remove it, 'q' means to quit, 'g' means to go and remove
   the rest of the named files and any other response will result in the file
   not being removed.

   -q quit if the file doesn't exist or if the remove fails for any reason.

   -v verbose. Rm will write the name of each file to stdout as it removes it.

SEE ALSO
   The UNIX command 'rm'

DIAGNOSTICS
   A message is printed if the file could not be removed.

AUTHOR
   Craig Leres.

BUGS/DEFICIENCIES
NAME
rmdir - remove directory

SYNOPSIS
rmdir [-dpsv] dir_name ...

DESCRIPTION
Rmdir is a tool to remove the specified directory, and all files/directories
contained in the specified directory.

If dir_name is empty, rmdir removes it. If dir_name is not empty, rmdir will
query with:

rmdir: Directory dir_name/ is not empty. Continue?

A response of "y" or "Y" results in the removal of dir_name (see below.) Any
other response causes rmdir to leave dir_name alone.

-d This flag prevents rmdir from confirming removal of non-empty direc-
tories. Instead, rmdir will attempt to remove all the files in the direc-
tory, and then remove the directory.

-v This flag causes rmdir to list the files and directories as they are
deleted.

-p This flag is like -v except the full path name of the deleted files and
directories are listed.

-s This flag causes rmdir to delete files using system protection, instead of
owner protection, which is the default. It is a fatal error for a user
without system privileges to use the -s flag. Users with a UIC-group
number between 1 and 10 octal (inclusive) are considered to have sys-
tem privileges.

SEE ALSO
Istree(tool), rm(tool)

DIAGNOSTICS
"rmdir: Name stack overflowed."

This error message means that there was a problem internal to rmdir. (Pos-
sibly dir_name had too many files/directories in it.)

"rmdir: Failed to delete file - <file>"
"rmdir: Failed to delete directory - dir_name/"

These errors may occur when the named file or directory does not belong to
the user.
"rmdir: Couldn't get file protection - <file>"
"rmdir: Couldn't set file protection - <file>"

The named file could not be accessed for reading the file protection or for changing the file protection.

AUTHOR
raig Leres

BUGS/DEFICIENCIES
If rmdir finds a file or directory that it cannot remove, it aborts after removing some, but not all of the files.

System performance may be reduced when the entire file system is removed.
NAME
rmotd - read a modcomp tape

SYNOPSIS
rmotd [filenames...]

DESCRIPTION
This tool uses the tool mtr to read a tape written on the Modcomp by the tool 'wvaxt'. This tool will read files from the tape mounted on MTA0:, stopping only when three consecutive file marks are read. If no filenames are given, or not enough are given (nine is maximum), some names are supplied. As each file is written, the name of the file is echoed to ERROUT.

Before invoking this tool, the following dcl command must be executed from a parent process:

$ MOUNT /FOREIGN/DENSITY=800 MTA0:

(A parent process is one which spawns the rmodt tool.)

FILES

SEE ALSO
mtr(tool), mtw(tool) wmodt(tool) rvaxt(tool) and wvaxt(tool) on the Modcomp

AUTHOR
Bob Upshaw
NAME
rmodtr - read modcomp tree

SYNOPSIS
rmodtr

DESCRIPTION
Rmodtr reads a tape containing a directory tree which was written on the
modcomp. The directory tree on the tape is put in the current working direc­
tory.

To copy the directory /mnt/gronk on the modcomp to /mnt/foobar on the
vax, put a tape on /dev/mt on the modcomp, and type

    cd /mnt/gronk
    wvaxtr

Then take the tape to the vax, put it on the tape drive, and say

    cd /mnt/foobar
    rmodtr

If rmodtr is unable to create any of the directories that were in the directory
tree on the modcomp, it continues anyway.

If rmodtr is unable to write any of the files onto a file with the same name as
the file on the modcomp, it writes a message, and creates a file name. The
file names created are of the form 'mtrfile.nnn', for some digits 'nnn'.

Like cp, if rmodtr is asked to read a file into a file that already exists, if asks
the user to confirm. (See cp(tool).)

SEE ALSO
rmodt(tool), wvaxtr(tool) (on the modcomp)

AUTHOR
odd Hammond

BUGS/DEFICIENCIES
NAME

roff - format text

SYNOPSIS

roff [+n] [-n] [-s] [-r] [-pon|-po n] [name ...]

DESCRIPTION

+br 'Roff' formats text according to 'request lines' embedded in the text of
the given file names. If nonexistent file names are encountered they are
ignored. The optional flag arguments mean:

+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 "Position paper, then hit CR to
continue" is given just once before the first page.

-r Produces roff input onto standard output. Useful for turning
documents which use RTSG macros into standard roff
documents which can be ported to other non-RTSG
software tools machines.

-PO n Move the entire document 'n' spaces (default=0) to the right
('page offset'). A space is not required before between '-
po' and 'n'. However, a number 'n' must be specified after
'-po'.

Input consists of intermixed text lines, which contain information to be for­
matted, and request lines, which contain instructions about how to format
the text lines. Request lines begin with a distinguishing 'control character',
normally a period. Inside the text lines, the user may specify that a charac­
ter is to be passed through roff intact by preceeding it with an at sign, @. This feature is particularly useful for spaces, which are usually treated some­
what freely by roff. (Although strings of spaces are processed so that at least
as many spaces appear as in a string, if the user does not escape the spaces,
other spaces may be added.) A line having leading spaces is output as if it
were preceded by a '.br' command (see request summary).

Output lines may be 'filled' without regard to the format of the input text
lines. Right justification of filled lines may be turned on and off through the
use of the '.ju' and '.nj' request lines. Line 'breaks' may result from the use
of certain request lines, or by the appearance of an empty input line or an
input line beginning with a space.

Normally, at least one space is placed between the last word of a line and the
first word of the next.
The capabilities of roff are specified in the attached 'Request Summary'. Numerical values are denoted there by 'n', titles by 't', and single characters by 'c'. Numbers denoted 'n' 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 head and foot for this writeup could have been set by:

.he 'ROFF'24May80'ROFF'
.fi

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 is used as the 'delimiter' to separate the three fields. Any '#' characters in a title are replaced by the current page number. Any '%' characters in a title prints the current date.

Tabs in the input are expanded according to the tab stops set with the ".'ta" command. There are 3 possible justifications of text on a tab stop: left, centered and right. Left justification is the default, centered & right justification are indicated by a 'c' or 'r' immediately following the tab position. For example, the input

.nf
.ta 8 16r 24c
abc/def/ghi/jkl

(where '/' indicates a tab) would result in 'abc' starting in column 1, 'def' left justified against column 8, 'ghi' right justified against column 16 and 'jkl' centered around column 24. (the roff tabs were copied directly from nroff tabs - for more information, see the UNIX nroff manual).

It is possible to write macros in roff. It is easier to enter text with macros, and if you want to change the formats of many documents, you need only to change the definitions of the macros. f(tool), m(tool), and memo(tool) are three useful macro packages. See roffmac(tool) if you want to write your own macros.

FILES
None
SEE ALSO

DIAGNOSTICS
None

AUTHOR
This tool was obtained from the book "Software Tools" by Kernighan and Plauger; it was initially adapted to BKY (including the ' -n', ' +n', ' -s ' and ' -po n' flags) by Debbie Scherrer. Minor additions were made by Donn Davy and Dennis Hall. Roff was originally prepared for installation by Brad Heckman. Substantial enhancements were added by Joe Sventek. Many bugs in boldface and underlining were fixed by Jef Poskanzer. Tabs were added by Van Jacobson.

BUGS/DEFICIENCIES
Lines taken from roff input files which begin with '..' are recognized as comments. Any of those lines read while expanding a macro with the '-r' option will be deleted. However, any of those lines read while NOT expanding a macro will be passed through, generating a non-standard roff input file.
REQUEST SUMMARY

Request Init. Default Break Meaning

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.bd n</td>
<td>Boldface the next 'n' lines.</td>
</tr>
<tr>
<td>.bp n</td>
<td>Begin new page and number it 'n'.</td>
</tr>
<tr>
<td>.br</td>
<td>The filling of the current line is stopped.</td>
</tr>
<tr>
<td>.cc c</td>
<td>Control character becomes 'c'.</td>
</tr>
<tr>
<td>.ce n</td>
<td>Center the next 'n' input lines without filling.</td>
</tr>
<tr>
<td>.ef t</td>
<td>Foots on even pages are 't'.</td>
</tr>
<tr>
<td>.eh t</td>
<td>Heads on even pages are 't'.</td>
</tr>
<tr>
<td>.fi</td>
<td>Begin filling output lines.</td>
</tr>
<tr>
<td>.fo t</td>
<td>All foot titles are 't'.</td>
</tr>
<tr>
<td>.he t</td>
<td>All head titles are 't'.</td>
</tr>
<tr>
<td>.ig cc</td>
<td>Stops printing text until &quot;cc&quot; appears alone on a line.</td>
</tr>
<tr>
<td>.in n</td>
<td>Set left margin to column 'n+1'.</td>
</tr>
<tr>
<td>.ju</td>
<td>Right justify filled lines.</td>
</tr>
<tr>
<td>.ls n</td>
<td>Line spacing set to 'n' lines per output line.</td>
</tr>
<tr>
<td>.m1 n</td>
<td>Put 'n' blank lines between top of page and head.</td>
</tr>
<tr>
<td>.m2 n</td>
<td>Put 'n' blank lines between head and text.</td>
</tr>
<tr>
<td>.m3 n</td>
<td>Put 'n' blank lines between text and foot.</td>
</tr>
<tr>
<td>.m4 n</td>
<td>Put 'n' blank lines between foot and bottom of page.</td>
</tr>
<tr>
<td>.ne n</td>
<td>Begin new page if 'n' lines won't fit on output. Break if needed to start new page; don't if not.</td>
</tr>
<tr>
<td>.nf</td>
<td>Stop filling output lines.</td>
</tr>
<tr>
<td>.nj</td>
<td>Do not right justify filled lines.</td>
</tr>
<tr>
<td>.of t</td>
<td>Foots on odd pages are 't'.</td>
</tr>
<tr>
<td>.oh t</td>
<td>Heads on odd pages are 't'.</td>
</tr>
<tr>
<td>.pl n</td>
<td>Total page length is 'n' lines.</td>
</tr>
<tr>
<td>.po n</td>
<td>Page offset; precede all lines by 'n' spaces.</td>
</tr>
<tr>
<td>.rm n</td>
<td>Set right margin to column 'n'.</td>
</tr>
<tr>
<td>.sp n</td>
<td>Insert 'n' blank lines, except at top of page.</td>
</tr>
<tr>
<td>.ta p</td>
<td>Define tab stops a position(s) 'p'.</td>
</tr>
<tr>
<td>.ti n</td>
<td>Temporarily indent next output line 'n' spaces.</td>
</tr>
</tbody>
</table>
.ul n   n=1   no   Underline the next 'n' lines.
   Only letters and digits are
   underlined.
NAME
roff - macro processing for roff

SYNOPSIS
(see description)

DESCRIPTION
Though this is not the way macros in roff actually work, it is probably best to
think of them as being piped through a tool (called, for example, roffmac) to
roff.

The input to this 'roffmac' tool is a mixture of text and macro invocations.
After all the macro invocations are evaluated, the resulting text is sent to
roff, and is formatted.

A macro invocation is of the form

. name a-1,..a-k

followed by either a semicolon or by a newline. Note that the semicolon or
newline is used for punctuation, and is actually part of the macro call. After
the macro is evaluated, the semicolon or newline will disappear.

When the macro 'name' is evaluated, first a-1 through a-k are evaluated, then
the macro is called with what a-1 through a-k evaluate to as its first through
k'th arguments. a-1 through a-k may also contain macros.

There are several built in macros. In the following {string} represents what
'string' evaluates to. Also, all the macros will be terminated with a ';';
although it is better in practice to terminate them with a newline, just as
code tends to be more readable when there is just one statement per line.

{.define a,b;}

is the null string, and has the side effect of creating a macro with
name \{a\} and definition \{b\}.

{.incr a;}

is \{a\} + 1, were \{a\} is interpreted as a decimal integer.

{.substr a,b,c;}

is \{the substring of \{a\} starting at \{b\} and having length \{c\}\}. (If c is
not specified, if \{c\} is the null string, or if \{c\} is too big, the remainder
of the string is taken. If \{a\} larger than the string, the result is the
null string.)

{.ifelse a,b,c,d;}

is \{c\} if \{a\} equals \{b\}. Otherwise, it is \{d\}.

{.arith a,b,c;}

- 1 -
is \{a\} + \{c\}, \{a\} - \{c\}, \{a\} * \{c\}, or \{a\} / \{c\}, where \{b\} is '+', '-', '*', or '/' respectively.

If 'name' is any user-defined macro,

\{name a-1,...,a-k;\}

is \{s\}, where 's' is the definition of 'name' after making the substitution \{a-1\} for '$1', \{a-2\} for '$2', and so on. Any arguments a-k that aren't specified are assumed to be the null string. If c is a non-digit, '$c$' is replaced by 'c'. In particular, '$$' is replaced by '$'.

There are two ways of 'quoting' something to prevent it from being evaluated.

\{[string]\}

is string. 'string' may contain paired brackets. For example, \{[abc[def]ghi]\} = abc[def]ghi. (Brackets are not special outside of macro calls.)

The user can also quote a single character by preceding it with '@'. That is,

\{@c\}

is 'c'. There are two exceptions to this rule: \{@t\} is a tab character, and \{@<NEWLINE>\} is the null string.

\{c\} is c for any non-special character c (i.e., any character except a comma, semicolon, newline, period, bracket, or atsign).

SEE ALSO

roff(tool), macro(tool)

BUGS/DEFICIENCIES

There are several bugs. One of the bugs that causes the most trouble is that macro calls such as

\{name a,b,c,...\}

frequently do not work correctly if any of \{a\}, \{b\}, \{c\}, ... has an unescaped comma, semicolon, newline character, or other special character which is outside of a macro. For example, if \{a\} is 'x,x',

\{ifelse a,[x,x],y,z;\}

turns out to be 'x,x', even though it should be to 'y'. This is not at all an isolated case: most macros will not work in this sort of case. Sometimes, it is possible to solve the problem by having 'a' evaluate to something like 'x@,x', which works correctly.

The problems arising from the bug above could be almost eliminated if a macro called 'value' such that
\{.value a;\}

would be the definition assigned to the variable \{a\}. The implementation of this macro would be fairly straightforward.

There are also occasional problems when a macro is given arguments which evaluate to strings with unmatched brackets '[' or ']' or with dollar signs '\$' in them. Note that user defined macros turn '$x$' to 'x', for 'x' a non-digit, while non-user defined macros leave '$x$' as '$x$'.

It might be more useful if when macros are evaluated, if '$k$' (for k a digit) were replaced by a quoted version of the k'th argument.

Macros are evaluated inside comments. (Typical roff comments are lines starting with '..'.)

Even if a control character (set by .cc, and initially '.') in column one is escaped with an '@', it will still be seen as a control character.
NAME
rtsgmac - macros to format text

SYNOPSIS
(see attached command summary)

DESCRIPTION
Rtsgmac is a set of roff macros used to simplify the production of documents. It is generally used by the tool f (format document). It contains macros to layout title pages (including "standard" headers and footers for each page), macros to automatically number sections, sub-sections, chapters, etc. of a document and macros to do paragraphing and various types of "displays". The different types of macros are independent. I.e., the paragraphing macros can be used even if the titling and section numbering macros aren't.

FILES
/usr/lib/rtsgmac (the source of the macros)

SEE ALSO
f(tool), roff(tool), manmac(lib) st.2.2.5 - "Macros for Standard Document Preparation"
COMMAND SUMMARY

The following is a list of the commands available with the current macro package.

Numbered Sections

[ .s <section title>] next section at current level. E.g., from "2.1" to "2.2".

[ .sn <section title>] first section at next level. E.g., from "2.1" to "2.1.1".

[ .sl <section title>] next section at previous level. E.g., from "2.1.1" to "2.2".

[ .nextlevel] down one section level (no print)

[ .lastlevel] up one section level (no print)

Paragraphs

[ .pp] normal paragraph (1st line indented 4 spaces).

[ .lp] left adjusted paragraph (no indentation on 1st line).

[ .ip <tag> ] indented paragraph with right-adjusted hanging tag

[ .tp <tag> ] same as .ip but with left-adjusted hanging tag.

[ .np] numbered paragraph. numbering begins at 1 and is incremented automatically on every use of .np until reset by a .lp, .pp, etc.

Displays

[ .ds] start display

[ .de] end display

Initialization and Titles

[ .title <document title>] setup title for .start

[ .draft] put "DRAFT" in the footer of every page.
[ .author <author>] setup author(s) for .start
[ .number <doc number>] setup document number for .start
[ .version <version>,<date>] setup version number for .start
[ .start] create title page, set margins, headers and footers.
[ .setmargins] setup margins without making title page or headers or footers.
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. The default number of columns is 79.

FILES

SEE ALSO
field(tool) - utility for field manipulation
sort(tool) - file sorter

DIAGNOSTICS
Passing a negative numeric argument will cause the tool to blow up like mad.

AUTHOR
Joe Sventek

BUGS/DEFICIENCIES
The maximum ruler size is 137 columns.
NAME
sched - repetitively invoke a shell command

SYNOPSIS
sched [-r<n>] [-t<n>] "command"
- r: repeat command <n> times
- t: wait <n> seconds between repeats

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 -r360 -t10 "ps -a >>ps.log"&

would cause ps to run and append its output to ps.log every 10 seconds for
the next hour. The resulting log file could then be edited to find out the most
heavily used utilities, the average number of processes per user, etc.

FILES

SEE ALSO
sleep(lib)

DIAGNOSTICS

AUTHOR
Joe Sventek

BUGS/DEFICIENCIES
NAME
  sedit - stream editor

SYNOPSIS
  sedit [-n] [[-e script | -f sfile]... | script] [file]...

DESCRIPTION
  Sedit copies the named input files to the standard output, performing editing
  as directed by sedit commands in "script" or in "sfile". The -e flag indicates
  that the next argument is to be interpreted as a sedit command (see below).
  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 inter­
  mixed 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 a sedit command. Normally, sedit writes each line of input to the output
  after editing; the -n option suppresses this action. As a result, the only out­
  put is that resulting from sedit commands.

  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 edit). 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 com­
  mand 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.

  Sedit accepts the following commands. Each command may be used with 0,
  1, or 2 line numbers.

(A) Whole-line Oriented Commands

    a
    <text>

      Append. The <text> is placed on the output after each selected
line. The \text{<text>} does not change the line number nor is it subject to subsequent sedit commands.

\text{c} \quad \text{<text>}

Change. The selected lines are deleted and \text{<text>} is placed on the output in their place. The \text{<text>} does not change the line number nor is it subject to subsequent sedit commands.

\text{d}

Delete. The selected lines are deleted.

\text{i} \quad \text{<text>}

Insert. The \text{<text>} is placed on the output before each selected line. The \text{<text>} does not change the line number nor is it subject to subsequent sedit commands.

\text{p}

Print. The selected lines are printed on the standard output.

\text{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{<text>} in an a command. The new lines do not change the line number nor are they subject to subsequent sedit commands.

\text{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.

\text{(C) Substitution Command}

\text{s/pat/new/gp}

Substitute. The leftmost occurrences of pat in the selected lines is changed to new. If g is specified, all occurrences are changed. If p is specified, the resulting line is printed.

\text{(D) Flow-of-Control Command}

\{ (command)
<commands>
}

Grouping. This command do no editing on the input lines, but control the application of commands to the lines selected by the address part. The grouping command '{' causes the next set of commands to be applied as a block to the input lines selected. The first of the commands under control of the grouping may appear on the same line as the '{' or on the next line. The grouping of the commands is terminated by a matching '}' standing on a line by itself.

(E) Miscellaneous Commands

= Print line number. The current line number is printed on the output as a line.

The a, c, i, and grouping commands may not appear in command line scripts.

Text appended by a, c, or r commands is placed on the output in the same order as the execution of the commands. Similar comments apply to text inserted by i commands.

Sedit can accommodate commands totaling approximately 5000 characters (including <text> arguments), and lines up to 120 characters in length.

SEE ALSO
edit(tool), change(tool), find(tool), tr(tool)

DIAGNOSTICS
In addition to the usual error messages resulting from file access failure, sedit issues the following messages preceding by the offending command line.

bad line numbers
indicates that the line number expressions are invalid.

invalid command
indicates that the command preceding 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.
AUTHOR
Whei-Ling Chang

BUGS/DEFICIENCIES
NAME
setenv - set the value for an environment variable

SYNOPSIS
setenv name value

DESCRIPTION
setenv is a built-in shell command which sets or defines environment variables. Environment variables are case-insensitive. Any variable "name" desired can be set using setenv.

Both "name" and "value" are single ascii strings, optionally enclosed in quotes. Variable names are case-insensitive, i.e. "PATH" and "path" name the same environment variable. The name and value pair are stored in the user"s "environment". If the value of an environment variable consists of multiple parts, it must be enclosed in quotes. For example;

```
% setenv PATH "~/bin /usr/bin /etc/bin"
```

Setenv will store the above value for "PATH" as a single string consisting of the characters between the quotes. No other attempt is made to interpret the value string. A "value" must be specified. However, null or empty values are allowed, i.e.

```
% setenv VERBOSE ""
```

thus allowing the implementation of "toggle" variables, i.e the value isn't important, just whether the variable is set or not.

AUTHOR
Theresa Breckon

SEE ALSO
printenv(tool), resetenv(tool), unsetenv(tool), envlb(lib), sh(tool)

DIAGNOSTICS
"setenv: Couldn't enter variable 'xxx' into environment." - An error occurred while trying to enter the variable xxx into the environment. This error should be reported to your software manager.
NAME
  sh - shell (command interpreter with history functions)

SYNOPSIS
  sh -anvx script [arg1 ... [argn]]

  or

  sh [-anvx] [-c argline]

DESCRIPTION
  Sh interprets command lines. It uses the history library (hislib(lib)) to
  implement a history mechanism. Sh can execute programs, or collections of
  other command lines.

  'Script' is the name of the file to read commands from. The shell uses the
  +PATH environment variable to locate 'script' if necessary. 'Arg1' through
  'argn' are the arguments to be to be given to the command file 'name'. If the
  'script' argument is omitted, sh reads the commands from standard input.

  The following flags are meaningful:

    -x echo lines as the shell reads them

    -v report what the shell is doing as it is doing it

    -n process the commands given, but don't actually execute them

    -a don't abort if some tool can't be spawned, or if some tool or script
      spawned by the shell aborts. Instead, continue as if the tool was spawned
      and didn't abort.

  Normally, if the shell is reading its commands from something other
  than a terminal, it will abort in either of these cases. No further tools or
  scripts referenced in the script are spawned, even if they are on the
  same command line as the tool which aborted or couldn't be spawned. See sh(tutorial) for more details.

    -c the next argument is a command line: the shell will only execute this
      command line, and will not read commands from standard input

When the shell is ready to execute a command from the terminal it prompts
the user for the command with a '\x1b[?1036a'. The user can then type the command,
terminating it with a carriage return. If the user can not finish typing the
command on one line, he can type an un-escaped 'at' sign (@) and a carriage
return, and the shell will prompt the user to continue typing the rest of the
command. ('Escaped' is explained later.)
Command lines given to the shell which start with a '#' character (possibly preceded by one or more spaces), are assumed to be comments, and are ignored.

The most basic command you can give to the shell is

```
program arg1 ... argn
```

This will execute the program which is in the file 'program' with the arguments arg1 through argn. Individual arguments are separated by spaces.

If the full pathname of 'program' is not given explicitly, the shell uses the PATH environment variable to locate it. See files(info) and paths(info).

If you want to put spaces or other characters which are special to the shell in an argument, you must 'escape' the argument:

To tell the shell not to interpret any characters in an argument as one of the shell's special characters, you can surround an entire argument by single quotes ('') or by double quotes ("').

To tell the shell that some given character (outside of quotes) is part of an argument, and shouldn't be misinterpreted as a special shell character you can precede it by an at sign, @.

An at sign followed by certain characters, however, may give you a special character that is hard to type in instead of the character itself:

```
@b = backspace
@t = tab
@d = delete
@@ = @ (at sign)
```

etc.

See esc(lib) for a complete list.

Any of these forms of arguments can be concatenated with another of these forms of argument to make a larger argument.

A collection of shell commands can be put in a file, and then you can execute it just the way you execute a program (i.e. give the shell the name of the file (called a script file) and some arguments).

In a command in a script file, any occurrence of $n outside of single quotes is replaced by the n'th argument for 0 < n.

Generally, most tools read from the user's terminal, and write to the user's terminal. It is possible, however, to have the tools read or write to or from some file other than the terminal. The tool's primary input is called 'standard input', the tool's primary output is called 'standard output', and the
place where the tool writes most of its error messages is called 'error output'. Many tools read and write to files other than these as well. You should see the manual entry for the tool to see exactly when reads and writes to and from standard input and output, and when it reads and writes to and from other files.

To change the default definitions of the standard files from the terminal, use the following conventions:

```bash
change standard input
command < infile

change standard output:
command > outfile
command >> outfile (for append access)

change error output:
command 1 outfile
command 2 outfile (for append access)
```

Inside a script, the user may want to specify some particular data to be data to a command. The way he does this is by using a 'here document':

```bash
command << 
data
```

This will cause the data 'data ...' to be the information in the standard input for the command 'command'. The character specified ('~' is used above) may be any printing ascii character. (It may have to be escaped on the command line, however.) $n parameters are evaluated inside of here documents, unless the character is preceeded by an atsign (right after the "<<").

When multiple here documents are used in a command, the data for the here documents must follow the entire command, and here documents are associated with the command as the command is read from left to right. An example will make this clearer:

```bash
command1 << ; command2 << = data for command1
~
data for command2
```

If 'command1' and 'command2' are commands,

```bash
command1 | command2
```
is a command which executes command1, and uses the output from that command as the input to command2.

If 'command1' and 'command2' are commands,

\[
\text{command1 ; command2}
\]

is a command which executes command1, and then executes command2.

If 'command' is a command, then

\[
\text{command &}
\]

is a command which executes 'command' as a separate process, and returns immediately. When 'command &' is executed, the shell types a number to the user's terminal. This number may later be used to abort the command 'command &'.

Parentheses can be used to resolve ambiguity in interpreting commands. For example,

\[
( \text{command1 ; command2 } ) \mid \text{command3}
\]

is a command which combines the output of command1 and the output of command2, and uses that as input to command3.

The shell executes certain commands directly. See the corresponding manual entries (section 1) for information on these commands.

\[
\begin{align*}
\text{cd} & \quad \text{change working directory} \\
\text{if} & \quad \text{execute a command, if some condition is true} \\
\text{goto} & \quad \text{resume execution of a script at the point specified by a label. A label in a script is a line which starts with ':'}, \text{ followed by one or more spaces, followed by the label name. Label lines are ignored except on gotos.} \\
\text{exit} & \quad \text{terminate the execution of a shell scripts} \\
\text{logout} & \quad \text{exit the shell}
\end{align*}
\]

The user can exit the shell by typing an end of file, as well as by typing 'logout'.

See also the manual entries for the environment variable built-in commands.
SEE ALSO
   files(info) and paths(info)
   sh(tutorial)
   hisib(lib) - the history library.
   cd(tool), if(tool), goto(tool), exit(tool), logout(tool)
   printenv(tool), setenv(tool), unsetenv(tool), resetenv(tool)
   envlb(lib) - environment library.

DIAGNOSTICS

<tool>: not found
   the program or script <tool> was not found. +PATH is the environment
   variable the shell uses as a search path to try to find the tool.

Various other error messages are printed when unsyntactical commands are
given to the shell.

AUTHOR
   Designed by Van Jacobson, Vern Paxson, and Marshall Spight. Implemented
   by Marshall Spight.

BUGS/DEFICIENCIES
NAME
sleep - cause a process to suspend itself for awhile

SYNOPSIS
sleep n

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(tool) - a way to repetitively invoke a command

DIAGNOSTICS

AUTHOR
Joe Sventek

BUGS/DEFICIENCIES
NAME
sort - sort and/or merge text files

SYNOPSIS
sort [-bdfimrn] [-tc] [+n[.m][-j[.k]][bdifnr] ...] [file ...]

DESCRIPTION
Sort sorts and/or merges 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
be used as a filter.

The default sort key is the entire line. The default ordering is alphabetic by
characters in the ASCII collating sequence. One flag, '-m', says that the files
are already sorted and only need to be merged. The merging is done based
on whatever key specification you supply.

Several flags affect the collating sequence:

  '-b' ignores leading blanks and tabs.

  '-i' ignores non-printing characters (ASCII control characters).

  '-d' ignores all characters but letters and digits. (If sorting in lexical
  order, this results in a "dictionary" order sort, hence the name of the
  flag).

  '-f' folds all letters to lower case before comparing.

  does a numeric sort. The keys are interpreted as real numbers. If the
  field starts with a non-numeric character, it collates as if it had the
  value '0.0'. Leading blanks and tabs are always ignored but other char­
  acters are affected by the flags above. E.g., "1?2" is treated as "1" under
  '-n' but as "12" under '-dn'. (I know this is weird - send a letter to Brian
  Kernigan).

  '-r' reverses the order of the sort. Normal order is least key first.
  This flag causes the order to be greatest key first.

The options following a '-' are "global". I.e., they apply to each following key
spec unless they are overridden by putting options after the key spec (this
may make more sense after the description of key specs below). If no
options are supplied, the defaults are a lexical sort with all characters
significant.

The default for sort is to use the entire line as a single key. However, the
sort can be restricted to part of the line and/or parts of the line can be used
as secondary keys if key specifications are given. Sort recognizes fields
separated by a field delimitter. (the default field delimitter is a TAB but it
can be changed to any character 'c' with the '-tc' flag). A key specification consists of start and end positions for the key and an optional new ordering for the key. The general form of the key spec is:

+<start><end><flags>

where <start> is the starting position of the key, <end> is the 1st character after the end of the key and <flags> are the same as the key flags described above. Both <start> and <end> have the form 'field.offset' where <field> is the field number (the 1st field is numbered 0) and <offset> is a character offset into the field (the 1st character of the field is numbered 0). The <offset> part (and its leading '.') may be omitted, in which case the offset is taken as 0. The entire <end> spec (together with its leading '-') may be omitted, in which case the end is taken as the start field + 1 with an offset of 0. If the <flags> are omitted, the "global" flags are used.

Some examples of the use of sort are:

sort x sorts "x" in lexical order
sort -n x sorts "x" in numeric order
sort -n +1 x sorts "x" in numeric order on its 2nd field
sort +0n +1-1.10 x sorts "x" in numeric order on its 1st field, then in lexical order on the 1st 10 characters of the 2nd field.

The sort algorithm is not 'stable': lines with equal keys do not retain their original input order.

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.

AUTHOR

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. Vern Paxson did much re-writing to get the external merge code to (a) be transportable and (b) work. Van Jacobson added numeric keys and sorting by sub-fields.
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.

It would be nice to have the "-u" flag of Unix sort. This flag deletes all but the first of all lines that compare equal. The tool `uniq(tool)` will do this if the lines are complete duplicates but uniq can't be restricted to some sub-field of a line.
NAME
spell - find spelling errors

SYNOPSIS
spell file

DESCRIPTION
Spell collects the words from the named file, folds all characters to a single case, and looks the words up in a dictionary. Any words not found in the dictionary are printed.

FILES
A dictionary file.

SEE ALSO
The Unix command 'spell'

DIAGNOSTICS

AUTHOR
Original version by J. Sventek

BUGS/DEFICIENCIES
The current version of this tool is implemented as a script file. It reads one and only one named file. Because of the mapping into lower case, words may be hard to locate in the original text. It is hoped a better version of 'spell' will be written in the future.
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 a 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.

AUTHOR
   David Hanson and friends (University of Arizona)

BUGS/DEFICIENCIES
NAME
Stmail - utility for sending mail to other users

SYNOPSIS
stmail

DESCRIPTION

Stmail is the utility used to send mail to other users of the system. Each user on the system has a file "mymail" that holds his unread mail. Stmail adds arriving messages to the end of each user's "mymail" files.

When user sends a mail, stmail will prompt for input of a list of To addresses and the body of the message. The list of To addresses should consist of user-names separated by commas. After input the To field, stmail will directly put user into the message input mode. When you reach the end of the message, type a "." character alone on the line. This will cause stmail send message to the delivery system and leave from the mail system.

While typing in a message to be sent to others, it is often useful to be able to invoke the text editor on the partial message, print the message, execute a shell command, or do some other auxiliary function. Stmail provides these capabilities through tilde escapes, which has a tilde (~) at the beginning of a line, followed by a single character which indicates the function to be performed.

Valid commands are:

- `~?` - to display this help list
- `~! command` - execute shell command
- `~b address(es)` - add address(es) to Bcc: field
- `~c address(es)` - add address(es) to cc: field
- `~e [editor]` - edit the message using 'editor'
- `~p` - print the message so far
- `~r file` - read file into message
- `~s text string` - add text string to Subject: field
- `~t address(es)` - add address(es) to To: field
- `~w file` - write message on file
- `~~` - begin a line with a '~'

Any other input results in an error message.

More detailed information of the stmail are described in msgtut (tutorial of msg).
FILES

SEE ALSO

msgtut - the msg tutorial
msg - the utility for sending and receiving mail
ed - the text editor
users - display valid mail recipients

AUTHOR

Whei-Ling Chang.

BUGS/DEFICIENCIES
NAME
strings - find the strings in a file

SYNOPSIS
strings [-n] [file] ...

DESCRIPTION
Strings reads its file arguments (or the standard input if none is given) and
sends to the standard output all the sequences of printable characters that
end in NEWLINE or EOS, if they are longer than a given threshold. The
optional argument "-n", where n is a digit strings, specifies the threshold. The
default threshold is four.

Strings is useful for identifying binary or otherwise confusing files.

FILES
None

SEE ALSO
dspc(tool)

AUTHOR
Marshall Spight

BUGS/DEFICIENCIES
Enormously long strings are subject to truncation.

The algorithm used for identifying strings is extremely primitive.

On the Modcomp, strings is limited to displaying ratfor strings.
NAME

stty - set terminal characteristics

SYNOPSIS

stty [option ...]

DESCRIPTION

Stty sets the specified terminal characteristics options. With no argument, stty will display the current characteristics of the terminal. The options are as follows:

crterase terminal is a CRT
-crterase terminal is a hardcopy
echo echo every typed character
-echo do not echo typed characters
eightbit send and receive eight bit data
-eightbit send and receive seven bit data
ff terminal can handle form feed character
-ff terminal can not handle form feed character
tabs terminal can handle tabs
-tabs terminal can not handle tabs
ucase map lower case characters into upper case
-ucase do not map lower case into upper case
widthN set the line length of the terminal to N.
wrap terminal automatically wraps at the end of a line
-wrap terminal does not wrap at the end of a line

50 75 110 134 150 300 600 1200 1800 2000 2400 3600 4800 9600 19200
set the terminal baud rate to the specified value
DIAGNOSTICS

stty: couldn't set terminal characteristics
stty: can't get terminal characteristics
stty: can't sense terminal characteristics
failed to get or set the terminal characteristics

stty: error in setting terminal speed
the terminal speed specified is not valid

stty: can't assign terminal channel
failed to assign a channel to the terminal

stty: can't deassign terminal
failed to deassign the terminal channel

AUTHOR
Dan Ruderman

BUGS/DEFICIENCIES
options may not be abbreviated
NAME
submit - submit shell commands to a batch queue.

SYNOPSIS
submit [-rk] [-q <qname>] shellcmds
   -r: don't report completion of job
   -k: keep DCL log file
   -q: use queue <qname>

DESCRIPTION
The submit tool lets a user specify a shell command line, complete with
pipes, redirects, etc., and have that command line executed in background
at low priority. No special privilege is needed to run submit.

Submit concatenates its arguments to form the shell command to be exe-
cuted. That means that pipes, redirects, and semicolons should be quoted,
or they will probably not do what is desired.

Submit uses +PATH/sh as the command line interpreter. Files are handled
relative to the directory from which submit was invoked, regardless of
whether or not the user subsequently changes directories. It is legal to put
"cd" commands in the list of commands to be executed.

The batch queue used is SYS$BATCH, unless the user specifies otherwise with
the -q flag.

A list of running jobs can be obtained with the DCL command "show queue
/all/batch".

SEE ALSO
sh(tool), Unix submit(1)

AUTHOR
Marshall Spight
NAME
suggest - make an on-line suggestion

SYNOPSIS
suggest [message]

DESCRIPTION
Suggest provides a mechanism for sending ideas to an on-line suggestion box. If used with an argument, suggest sends the argument (up to twenty-five words) as the text of the suggestion. With no arguments, it reads from standard input, prompting for a reply as with mail. It is also possible to use the same flags as mail to edit and read a file from standard input.

For example,

    suggest 'Why don't we keep frosted glasses in the freezer?'

Suggest sends the above message to a suggestion box which is read weekly and distributed at the Central Support Meetings. When writing in a suggestion, enclose the suggestion in quotes or exclude the use of special characters.

DIAGNOSTICS

SEE ALSO
mail(tool)

AUTHOR
Nancy Deerinck, Van Jacobson

BUGS/DEFICIENCIES
NAME
suspnd - suspend a running process

SYNOPSIS
suspnd processid [processid ...]

DESCRIPTION
Suspnd suspends running processes specified by the processid's in the com­
mmand line. The processid's are those returned by the shell when it spawns a
background process.

FILES
None

SEE ALSO
sh(tool) - shell (command line interpreter) resume(tool) - resume a
suspended process

DIAGNOSTICS
If the process cannot be suspended, an error message is displayed on error
output.

AUTHOR
Joe Sventek (VAX)

BUGS/DEFICIENCIES
NAME
tail - print last lines of a file

SYNOPSIS
tail [-<n>] [-] [file....]
   -<n>: print <n> lines
   -: read input from STDIN

DESCRIPTION
Tail prints the last "n" lines of the indicated file. If "n" is omitted, the size of
the user's terminal is used.

If "file" is omitted or is "-", tail reads from standard input.

SEE ALSO
   split(tool)

AUTHOR
   Marylou Orayani

BUGS/DEFICIENCIES
   Tail does not work if one of the files given is non-random access, e.g., a termi-
   nal.
NAME
tbl - tool to format tables

SYNOPSIS
tbl [-t<tabchar>] [-box] [-dbox] [-f formatfile] ...
[formatline] ...

DESCRIPTION
Tbl is a tool for producing nicely formatted tables from a file of data and a
simple description of the table layout. Even complex tables are relatively
eyou to specify and enter. Tables are made up of columns which may be
independently centered, right-adjusted, left-adjusted or aligned by decimal
points. Headings may be placed over single columns or groups of columns.
Horizontal or vertical lines may be drawn as desired in the table and any
table or element may be enclosed in a box.

Tbl is an almost straight copy of the UNIX program "tbl". The UNIX tbl
document (Tbl - A Program to Format Tables by M.E.Lesk) contains more
description and more examples than this manual entry and should be read
by anyone unfamiliar with tbl (a summary of the differences between this tbl
and the UNIX tbl is given in the last section).

1. An Example

The following example illustrates a typical use of tbl (and helps give some
perspective to the tbl commands described in the next section). Say that
you have some data on the composition of foods that you would like formatted
as a table. The command:

% tbl lrnn

would instruct tbl to format a table of 4 columns, one left adjusted and the
remaining 3 numeric (the character "~" in all of the examples represents a
TAB character). The tbl output would look like:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Apples</td>
<td>.4</td>
<td>13.0</td>
</tr>
<tr>
<td>Halibut</td>
<td>18.4</td>
<td>5.2</td>
</tr>
<tr>
<td>Lima Beans</td>
<td>7.5</td>
<td>8</td>
</tr>
<tr>
<td>Milk</td>
<td>3.3</td>
<td>4.0</td>
</tr>
<tr>
<td>Mushrooms</td>
<td>3.5</td>
<td>6.0</td>
</tr>
<tr>
<td>Rye Bread</td>
<td>9.0</td>
<td>52.7</td>
</tr>
</tbody>
</table>

If, instead, you would like the table in a box with lines separating the
columns you could use the command:
If you want centered headings over each of the columns, you could add:

```
% tbl -box "c|c|c|c" "c|c|c|c" "l|n|n|n"
Food|Protein|Fat|Carbohydrate
---|-------|---|--------------
Apples| .4   | .5 | 13.0
Halibut| 18.4 | 5.2 | *
Lima Beans| 7.5  | .8 | 22.0
Milk| 3.3  | 4.0 | 5.0
Mushrooms| 3.5  | .4 | 6.0
Rye Bread| 9.0  | .6 | 52.7
```

which results in:

If you think this would look better with the numeric columns all the same width, you could do:

```
% tbl -dbox "c|c|c|c" "c|c|c|c" "l|n|n|n|n" (same data as above)
```

which would result in:
<table>
<thead>
<tr>
<th>Food</th>
<th>Protein</th>
<th>Fat</th>
<th>Carbohydrate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apples</td>
<td>.4</td>
<td>.5</td>
<td>13.0</td>
</tr>
<tr>
<td>Halibut</td>
<td>18.4</td>
<td>5.2</td>
<td>*</td>
</tr>
<tr>
<td>Lima Beans</td>
<td>7.5</td>
<td>.8</td>
<td>22.0</td>
</tr>
<tr>
<td>Milk</td>
<td>3.3</td>
<td>4.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Mushrooms</td>
<td>3.5</td>
<td>.4</td>
<td>6.0</td>
</tr>
<tr>
<td>Rye Bread</td>
<td>9.0</td>
<td>.6</td>
<td>52.7</td>
</tr>
</tbody>
</table>

Bells and whistles can be added until you end up with something like:

<table>
<thead>
<tr>
<th>Composition of Foods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Food</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>----------------------</td>
</tr>
<tr>
<td>Apples</td>
</tr>
<tr>
<td>Halibut</td>
</tr>
<tr>
<td>Lima Beans</td>
</tr>
<tr>
<td>Milk</td>
</tr>
<tr>
<td>Mushrooms</td>
</tr>
<tr>
<td>Rye Bread</td>
</tr>
</tbody>
</table>

which comes from using the format:

css
ccss
ccss
cc|c|c|c
c|c|c|c
dl|ne|ne|ne

on the input:

```
Composition of Foods

~Percent by Weight
Food~
~Protein~Fat~Carbo-
~hydrate

Apples~.4~.5~13.0
Halibut~18.4~5.2~*
Lima Beans~7.5~.8~22.0
Milk~3.3~4.0~5.0
Mushrooms~3.5~.4~6.0
Rye Bread~9.0~.6~52.7
```
2. TBL Commands

TBL needs two kinds of information to create a table: layout or format info (which comes from the arguments or a format file) and the data to format (which comes from standard input).

Data mostly consists of lines of fields separated by tabs. Each line will turn into one row of the table with each field in one column of the row. Fields may be empty. Trailing fields may be omitted and excess fields are ignored. The data may also cause horizontal lines to be drawn thru the table: a data line consisting of just "=" or "~" will cause a line to be drawn the full width of the table. A field consisting of just "=" or "~" will draw a line the width of the field. TBL will always make a column wide enough to accommodate the widest data field of that column (TBL makes 2 passes over the input file: one to compute the column widths and one to format the table).

Formats consist of lines of column types. The number of column types in the format line fixes the number of columns in the table (each format line must have the same number of columns). If column types on a format line are separated by a vertical bar, a vertical line is drawn between the corresponding data columns. Each format line is used for exactly one input line (except the last format is reused if there are more input lines than formats).

Each column type consists of a letter giving the type of justification for the column and optional spacing and width for the column. I.e.,

l gives a left justified column

r gives a right justified column

c gives a centered column

n gives a numeric column (numbers are justified with the units digits lining up vertically).

s "spans" the previous column over this column (this is how headings which span multiple columns are made).

The letters l, r, c or n may be followed by a number giving the amount of blank space to be left between this column and the next (the default is 3 spaces between columns). The spacing may be followed by the letter 'w' and a number giving the minimum width of the column (the column will be made wider than this if some data item requires it) or the letter 'e' (for "equal width" columns - all columns marked with an 'e' will be made the same width). E.g.,

is a centered column

is a centered column of data with 5 spaces between it and the next column
is a centered column at least 10 characters wide

is a centered column as wide as the widest of the "e" columns with 5 spaces between it and the next column.

Spaces are ignored anywhere in format lines (except in numbers) and letters may be in either upper or lower case.

FILES
/tmp/TOOxxx/tbxnnn is a temp file used if standard input is from a terminal.

SEE ALSO
roff(tool) "TBL - A Program for Formatting Tables" by M.E. Lesk

DIAGNOSTICS

AUTHOR
Van Jacobson

BUGS/DEFICIENCIES
If each format line doesn't contain the same number of formats (e.g., you say "tbl 'cs' 'nnn'") the output will look very strange.

In Unix tbl, formats and data are combined into one file. Unix tbl isn't stand-alone, its output must be piped thru nroff or troff.
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
cat(tool), crt(tool), UNIX tee

DIAGNOSTICS
A message is printed if the input file cannot be opened.

AUTHOR
David Hanson and friends (University of Arizona)

BUGS/DEFICIENCIES
NAME

tooldef - DCL command to set up the tools environment for DCL users

SYNOPSIS

$ @@rt_usr_,tooldef.com

DESCRIPTION

Executing the command procedure tooldef.com (in /usr) will assign symbols to (almost) every tool in /usr/bin. The tools not accessible to DCL users are those whose names conflict with DCL commands.

A system independent method to execute tooldef.com is given in the synopsis above. Note that the symbol "rt_usr_" is supplied as a system logical name.

FILES

/usr/tooldef.com
/usr/manager/tooldef.p - script to build tooldef.com

SEE ALSO

untooldef.com - Command procedure to remove the tools environment from DCL.

AUTHOR

Bob Upshaw
NAME

tr - character transliteration

SYNOPSIS

tr <infile >outfile 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 collapse 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

DIAGNOSTICS

"usage: translit from to."
   The command line passed to the tranlit is in error.
"from: too large."
   The string for "from" is too large. Current limit is 100 characters including EOS.
"to: too large."
   The string for "to" is too large. Current limit is 100 characters including EOS.

AUTHOR

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

BUGS/DEFICIENCIES
NAME
tsort - topologically sort symbols

SYNOPSIS
tsort [files ...]

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', 'c' 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

Note that get_out_of_bed DID NOT precede turn_off_alarm even though it did on the input line.

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(tool)

AUTHOR
David Hanson and friends (U. of Arizona)
NAME

typeof - determine the type of a file

SYNOPSIS

  typeof [-1 <n>] [file ...]
         -1: check <n> lines to determine type

DESCRIPTION

  Typeof performs a series of tests on each of the argument file(s) in an
  attempt to classify them. File types known to typeof are

     source
     binary
     uxl (MODCOMP ONLY)
     library (assembler or linker output)
     directory
     cant open

  If the file is a source file, typeof will read up to the specified number of lines
  (or 100 lines if not specified) and attempt to determine its language. Possibly
  source types are:

     ratfor
     fortran
     bcpl
     assembler-mc
     assembler-vax
     jcproc-mc (a Job Control procedure)
     jcproc-vax (DCL command file)
     insert (an assembler "INS" file-MODCOMP ONLY)
     roff (a Roff input file)
     unknown (not one of the above)

  If no arguments are given on the command line, file names are read from
  standard input. For example,

     lstree -p /usr/src | typeof

  will list the type of every program in /usr/src.

FILES

  None.

SEE ALSO

  gettyp(lib), typlb(lib), Unix file(1)
AUTHOR
  Van Jacobson, Jef Poskanzer

BUGS/DEFICIENCIES
  On the VAX, a library file is classified as binary. This may change in the future.
NAME
uniq - strip adjacent repeated lines from a file

SYNOPSIS
uniq [-c] [-i] [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 specified, uniq prepends to each output line a count of how many adjacent copies there were, using a tab to separate the count and the line. If the -i flag is specified, uniq ignores case differences in the input - "A" and "a" are treated the same.

FILES

SEE ALSO
comm(tool), UNIX uniq

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

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

BUGS/DEFICIENCIES
The Unix version of uniq contains several flags which are not yet implemented here.
NAME
unlock - return files or directories checked out

SYNOPSIS
unlock file ...

DESCRIPTION
Unlock returns files and directories that have previously been checked out with lock.

FILES
/</directory>/.lock - the file containing the list of files locked in that directory
/</directory>/.owner - the owner of the directory /</directory> (and particularly the owner of /</directory>/.lock)

SEE ALSO
lock(tool), lslock(tool)

DIAGNOSTICS

file not checked out to you <etc>
The user has tried to return a file that he didn't check out in the first place.

- <file>: unlocked
The lock program is able to unlock the file specified. It will actually be unlocked if there are no errors in unlocking other files.

- <files>: unable to unlock
The unlock tool can't unlock the file specified, probably because it was checked out to somebody else.

- <file>: not checked out
The user has tried to return a file that is not checked out at all.

- <file>: can't unlock - invalid file name
The user tried to unlock the file <file>, but there couldn't possibly be such a file, as the file name doesn't make sense.

- /: can't unlock root directory
The user has tried to unlock the root directory. The lock program doesn't know how to do this. He couldn't have locked the top level directory anyway.

- <file>: can unlock in only one project
The user has tried to unlock files contained in two different top level directories in the same command. He should unlock the files in separate commands.
**can't @re@open owner file**
Either somebody else is using lock or unlock at the same time as you are, and you should try again later, or you have tried to unlock a file in a toplevel directory that doesn't exist or doesn't have .lock and .owner files.

**can't @re@open lock file**
Somebody is looking at the lock file. Try again later.

**lock file garbaged**
The lock file is not in the proper format, and the unlock program is unable to process it. Send mail to trouble as soon as possible.

**fatal errors: no locks performed**
There were errors in attempting to unlock the files requested. *None* of the files that the user requested to unlock has been unlocked.

files specified have been unlocked
Everything's ok, and all the files requested to be unlocked have been unlocked. Nothing is unlocked unless the user gets this message.

**AUTHOR**
Todd Hammond

**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(tool), sort(tool)

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

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

BUGS/DEFICIENCIES
NAME
unsetenv - un-set the value for an environment variable

SYNOPSIS
unsetenv name [name] ...

DESCRIPTION
Unsetenv is a built-in shell command which unsets or undefines environment variables. Any variable "name" desired can be unset using unsetenv. Environment variable names are case-insensitive.

"Name" is an ascii string, optionally enclosed in quotes. Variable names are case-insensitive, i.e. "PATH" and "path" name the same environment variable. The specified names, and their values, are deleted from the user's "environment".

AUTHOR
Theresa Breckon

SEE ALSO
printenv, resetenv, setenv, envlib, sh

DIAGNOSTICS
"unsetenv: Couldn't remove variable "xxx" from environment." - Error occurred while trying to remove the variable xxx from the environment. This error should be reported to your software manager.
NAME
untooldef - DCL command to remove the tools environment from DCL

SYNOPSIS
$ @@usr:_untooldef.com

DESCRIPTION
Executing the command procedure untooldef.com (in /usr) will remove all
global symbols assigned previously by tooldef.com.

A system independent method to execute untooldef.com is given in the
synopsis above. Note that the symbol usr_is supplied as a system assigned
symbol.

FILES
/usr/untooldef.com
/usr/proc/tooldef.p - script to build untooldef.com

SEE ALSO
   tooldef.com - Command procedure to add the tools
       environment to DCL

AUTHOR
   Bob Upshaw
NAME
usage - display usage of a tool

SYNOPSIS
usage tool...

DESCRIPTION
The manual entries for the tools specified are found using the search path
specified in the environment variable MAN. The name and synopsis entries
for each tool is written to standard output.

Usage uses the MAN search path to find a manual entry, not the PATH search
path. Therefore, the manual entry found by usage may not correspond to
the tool by the same name found by the shell. The user should keep his/her
MAN and PATH environment variables compatible.

DIAGNOSTICS
"usage: non-standard manual for x" means a "name" line couldn't be found in
the manual entry for x.

AUTHOR
Bob Upshaw

SEE ALSO
man(info)
NAME
Users - list valid mail users

SYNOPSIS
users [-{<n> | v}]

DESCRIPTION
Users lists the valid addresses on the current host to whom mail may be
sent. First, the user names found in ~msg/address are listed, followed by
the aliases found in ~msg/malias.

Specification of the '-v' flag causes any verbose information associated with
the name or alias to be displayed with the name as a comment string (see
the manual entry for stmail for more details).

In the non-verbose mode, the addresses are displayed in 5 columns if stan­
dard output is to a terminal, or 1 column if not. Specification of the '-<n>'
option forces the output to be displayed in '<n>' columns.

FILES
~msg/address, ~msg/malias

SEE ALSO
postmn - utility to check for the existence of mail
resolve - utility to query information concering mail users

AUTHOR
Joe Sventek. Modified by Wei-Ling Chang.

BUGS/DEFICIENCIES
NAME
   vi - use vi from the shell

SYNOPSIS
   vi ...

DESCRIPTION
   This tool allows use the Unix vi editor from within the shell. Any valid vi arguments may be used. File specifications that begin with '~' or '/' are assumed to be full pathnames and are converted to Eunice format.

   The Tools environment variable TERM should be set to a string that corresponds to a valid Unix terminal type. If TERM is not set, the tool attempts to figure out your terminal type from VMS.

   The environment variable EXINIT, if present, is used as the ex options string.
   The default vi binary that is used is:

   eun_usr:[usr.new]vi.exe

   The environment variable VIBINARY can be used to override this default.

   The environment variable TERMCP can be used to override the default termcap file.

SEE ALSO
   ed(tool), edt(tool) Unix ex and vi documentation.
NAME
Vmstost - converts the VMS mail to MSG message format

SYNOPSIS
vmstost infile outfile

DESCRIPTION
Vmstost converts the message body of VMS mail in "infile" to the MSG message format and output to the specified outfile. The outfile can be handled by the MSG mail system.

SEE ALSO

AUTHOR
Whei-Ling Chang

BUGS/DEFICIENCIES
NAME
wc - count lines, words, and characters in files

SYNOPSIS
wc [-tvlwc] [file] ...

DESCRIPTION
Wc prints the number of lines, words, and characters in the named files. The
default is all three. The filename "-" specifies the standard input. A total is
printed if the '-t' flag is specified. A "word" is any sequence of characters
delimited by spaces.

The options -l, -w, and -c specify, respectively, that only the line, word, or
character count be printed. For example,

```
w -lc foo
```

prints the number of lines and characters in "foo". The -v flag causes a head­
ing to be printed along with the numerical information. If no files are given,
w reads its standard input.

DIAGNOSTICS
name: can't open
Printed when an input file can't be opened; processing ceases

AUTHOR
David Hanson and friends (U. of Arizona).
who (tool)  March 25, 1985  who (tool)

NAME
who - show who's on and what they are doing

SYNOPSIS
who [-a]
   -a: show batch jobs too

DESCRIPTION
Who provides information about the users currently logged into the system. It produces a line of output for each user, ordered by terminal name. The information displayed includes terminal name, username, connect time for the terminal and the name or names of images running at the terminal.

The optional "-a" flag causes batch jobs to be included in the output. The batch job number appears instead of a terminal name.

The connect time consists of the number of days, hours and minutes that a terminal has been logged on the system. (Leading zeros are suppressed.)

If there is more than one image running at a terminal, it is usually the case that only the last image in the list is executing.

Sample output:

nty1: jef 5 msg
opa0: system 32 directory.exe
tta3: smith 6:57
ttb1: bobup 30 sh, stmail
ttc2: fieldtest 2:34 essaa.exe
ttd6: marshall 47 sh, sh, bc, basic.exe
tte7: superuser 53 sh, sh, rc, fortran.exe
ttf0: leres 2 who
tth2: van 23 csh.exe, sysline.exe
tth6: chang 1:37 sh, msg, stmail

In this example, user smith has been logged on terminal tta3: for six hours and fifty seven minutes and is not currently executing any images.

SEE ALSO
ps (tool)

DIAGNOSTICS
"who: Not installed with WORLD privilege"

This error message occurs when who is not installed properly. In order for non-privileged users to run who, it must be installed with WORLD privilege. When this error occurs, complain to the system manager.

"who: Unknown status from sys$getjpi() - 0x<hex number>"

This indicates that something unforeseen has occurred. If you get this
message, please report the status code to the system manager.

AUTHOR
Craig Leres
whois(tool)  March 25, 1985  whois(tool)

NAME
whois - print out information about users

SYNOPSIS
whois [-i] [-] [user] ...

DESCRIPTION
Whois is a tool that lists information about specified users.

The "-i" flag causes information about the current user to be displayed. The
"-" flag can be used to force whois to read lines from STDIN. Other argu­
ments are assumed to be usernames. When invoked with no arguments,
whois reads from names from STDIN. When reading from STDIN, input lines
that are empty or begin the hash character are ignored.

The information is listed in these four columns: username, uic, home direc­
tory, and real name. For example:

    % whois leres superuser system superetc
    leres [231,004]  /mnt/leres  Craig Leres
    superuser [011,001] /usr  Super User
    system [001,004]  /sys/sysmgr  System Manager
    superetc [012,001]  /etc  Etc

SEE ALSO
who(tool)

DIAGNOSTICS

    whois: no such user - <user>

The specified argument does not correspond to a valid username on the sys­
tem.

AUTHOR
Craig Leres
NAME
wmodt - write a modcomp tape

SYNOPSIS
wmodt filenames...

DESCRIPTION
This tool uses the tool mtw to write a tape to be read on the Modcomp by the
tool 'rvaxt'. This tool will write files to the tape mounted on MT AO:, terminat-
ing the last file with three file marks. As each file is written, the name of the
file is echoed to ERROUT.

Before invoking this tool, the following dcl command must be executed from
a parent process:

$ MOUNT /FOREIGN/DENSITY=800 MTA0:
(A parent process is one which spawns the wmodt tool.)

FILES

SEE ALSO
mtr(tool), mtw(tool) rmodt(tool) rvaxt(tool) and wvaxt(tool) on the Modcomp

AUTHOR
Bob Upshaw

BUGS/DEFICIENCIES
There is currently a limit of 20 arguments which can be passed to wmodt.
NAME
write - write to another user

SYNOPSIS
write username write tty

DESCRIPTION
Write copies standard input to the interactive terminal of another user. When first invoked, it sends the message:

Message from <login_name> (<real_name>) on <tty>: <hh:mm:ss>

to the indicated user. Then, if standard input is also an interactive device, write will write to standard output the line:

Connection to <tty:> complete - go ahead

One way communication continues until an end-of-file (~Z) is read on standard input, at which time the recipient of the message is sent the line:

Message completed. <hh:mm:ss>

Two way communication occurs when two users are writing to each other.

If the character '!' is found at the beginning of a line, write will spawn a shell with the remainder of the line as a command. If a line with only '!' on it is entered, write spawns an interactive shell. The user can return back to write by logging out of the shell or by typing an end-of-file (~Z). Write then writes a '!' to the terminal to indicate its return from the spawn.

The following protocol is suggested for using write: When writing to a user you should end each message with an o (upper or lower case letter 'oh') on a line by itself. This will cause an -over- to be displayed on the recipients terminal along with proper carriage positioning. Thus, the user being written can respond without confusing your message with his/her messages.

To prevent abuse of "intelligent" terminals, all control characters, except TAB and NEWLINE, are converted to visible sequences.

It is possible to disable messages from appearing on a terminal. To disable messages (including system messages!) one simply types, from DCL:

$ set term /nobroadcast

To enable broadcasts use:

$ set term /broadcast
SEE ALSO
who(tool), sh(tool) The UNIX write tool

AUTHOR
Vern Paxson

DIAGNOSTICS

"write: can't write to <user>" This message can appear for a number of reasons: the user specified is not logged in; the terminal specified is invalid; or write is not installed with the correct privileges.

BUGS/DEFICIENCIES

For upward-compatibility, write allows a second argument, which simply overrides the first.

If someone's username happens to be a valid terminal name too, then whenever the user is logged in you can't write to the terminal.

In order to write to other terminals, the write tool uses a system call which precedes and terminates the line being written with a newline. This causes write messages to appear only on every other line.
NAME
  xref - make a cross reference of symbols

SYNOPSIS
  xref [-f] [files ...]

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.

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.

AUTHOR
  David Hanson and friends (U. of Arizona)

BUGS/DEFICIENCIES
  There should be a means of suppressing "junk" symbols such as "the", "a",
  etc.
NAME
   yacc - translate Yacc program into Ratfor

SYNOPSIS
   yacc [-dl] [-s<n>] [file]
      -d: parser should run in debug mode
      -l: list grammar file to error output
      -s<n>: set parser stack size to <n>

DESCRIPTION
   Yacc is a tool which generates parsers. It takes as input a file which contains
   a description of a grammar and produces as output a Ratfor program which
   will parse the grammar. Yacc recognizes the following flags:

   -d enables the generated parser to run in debug mode. This means that while
     the while the generated parser is parsing, it will write to the error output a
     listing of the shifts and reduces being performed, along with the relevant
     parse stacks.

   -l produce a listing onto yacc's error output containing the grammar file's
     symbols and their definitions, the grammar productions, a vocabulary cross­
     reference, and state sets used to build the parse tables which drive the gen­
     erated parser.

   -s<n> allows the user to change the parser stack sizes. <n> is the specified
     size, for example, '-s50' would set the stack sizes to 50.

FILES
   /usr/incl/yytblc
   /usr/lib/yyplb - yacc-generated programs must be linked with this file.

SEE ALSO
   yacctut(tutorial), lrgen(tool), yacclr(tool), yyplb(lib) "Yacc: Yet Another
   Compiler-Compiler" by S.C. Johnson "LR - Automatic Parser Generator and
   LR(1) Parser" by C. Wetherall and A. Shannon.

AUTHOR
   Theresa Breckon. Rewritten by Vern Paxson.

BUGS/DEFICIENCIES
   The parse tables generated by lrgen are kept in common blocks in the
   include file yytblc. Since these common blocks must be included in code that
   is created before the parse tables are created, the size of the each of the
   tables is set to 1 in the common blocks. Lrgen then outputs the same com­
   mon blocks, with the correct size blocks. However, trying to debug any code
   that contains the 1-dimensioned tables will result in errors like "array
   bounds exceeded".
DIAGNOSTICS

The following messages can appear when running the generated parser:

*Illegal Language Construct* - a point in the parse was reached where no
SHIFT or REDUCE was possible.

*Tried to Associate Non-associating Operator* - a sentence of the form "e1
OP e2 OP e3" was recognized, where OP was defined as non-associating in
the grammar definition.
NAME
    yacc - translates yacc source to lrgen source

SYNOPSIS
    yacc [-l] [-p] [-e <n>] [-s sfile] gfile

DESCRIPTION
    Yaclr translates a yacc grammar specification into one accepted by the lrgen tool. If no input is given, or the filename '-' appears, the standard input will be read. The '-l' flag causes yaclr to output a control character with the lrgen grammar which signals the lrgen tool to produce a listing containing the following information:

    - the lrgen grammar symbols and their definitions.

    - the grammar productions in BNF format.

    - a vocabulary cross-reference.

    - the configuration sets used to build the LR parse tables which drive the generated LR parser, along with a listing of all the conflicts for each state, even though these conflicts may have been resolved by precedence and associativity declarations.

    The '-p' flag causes yaclr to output a different control character, which causes lrgen to set a flag in the generated parser to put it into debug mode. This means that while the generated parser is parsing, a listing of the shifts and reduces being performed will be written to error output, along with the relevant parse stacks.

    The '-e' flag is used to specify the stack sizes for the parser, for example, '-e50' would set the stack sizes to 50. If no stack size is set, a default size is used.

    The '-s' flag causes yaclr to output the semantic routine and the programs section to the file "sfile". Otherwise this code is output to standard output along with the lrgen grammar.

    A yacc grammar specification consists of three sections: declarations, grammar rules, and programs. The sections are separated by double percent "%%" marks. A full grammar specification would look like:

    declarations
    %%
    rules
    %%
    programs

    Blanks, tabs, and newlines are ignored, except that they may not appear in any nonterminal or terminal (token) names. Comments are allowed in the
specification, they begin with '#', as in ratfor. All special characters described here must be delimited by blanks or tabs.

The declarations section can contain any combination of token declarations and semantic declarations or definitions. Tokens must be declared in the declarations section using one of the four keywords; %token, %left, %right, or %nonassoc. See the manual entry, yc(tutorial), for further explanations of these keywords. Token names, as well as nonterminal names, may be of arbitrary length, and may be made up of letters, dot '.', underscore '_', and non-initial digits. Upper and lower case letters are not distinct. The token definitions, or numbers, can be chosen by yaclr or by the user.

To assign a token number to a token, the first appearance of the token name in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name. Names not defined by this mechanism will get a default token number, chosen by yaclr. These default numbers start at 258. Any literals (single characters enclosed in single quotes), found in the productions are categorized as terminals. Their default token number is the numerical value of the literal character in ascii. The atsign '@' is an escape character for literals. The following escaped literals are recognized:

'@n' newline
'@r' return
'@@' atsign
'@t' tab
'@b' backspace
'@f' formfeed
'@c' c, where c is any other character

Semantic declarations and definitions can also appear in the declarations section. They must be enclosed in the marks '%{ ' and '%}'. For example,

%{
  include symbols  #tools definitions
  integer ctoi   #converts char to integer
  integer getlin  #gets next line
%}

could be placed in the declarations section. These declarations and definitions are made known to all of the semantic actions specified in the rules section. Thus, in the above example, the definitions in the include file 'symbols' and the functions 'ctoi' and 'getlin' would be defined for all of the semantic actions specified. Yaclr's parser uses names beginning with 'yy'; the semantic code should avoid such names.

The rules section is made up of one or more grammar rules. A grammar rule has the form:
LHS : BODY ;

LHS represents a nonterminal name, and BODY represents a sequence of zero or more terminals, (tokens), and nonterminals. All grammar rules having the same left hand side must be grouped together using the vertical bar '|' :

A : B C D
   | E F
   | G

With each grammar rule, the user may associate a semantic action to be performed each time the rule is recognized by the parser. Each action must be surrounded by two lines, the first line containing the mark '%{', and the last line containing the mark '} '. These actions may return values and may obtain values from previous actions. An action is any legal ratfor statement and as such can do input and output, call subroutines and functions, and alter variables. For example,

A : YYY ZZZ
   %{#THIS IS AN ACTION
     i = i + 1
     call putdec( i, 10 )
   %}

To make an action return a value to the parser, set the pseudo-variable '$$' to the value. For example, if for some rule you only wanted to return the value 1, follow the rule with the action

   %{$$ = 1
   %}

To obtain values returned by previous actions, use the pseudo-variables '$$1', '$$2', ... which refer to the values returned by the parts of the right hand side of the rule, from left to right. Thus, if the rule is

   expression : term '+' factor
   %{$$ = $$1 + $$3
   %}

then '$$1' has the value returned by the semantic action for 'term', and '$$3' for 'factor'. The value returned for 'expression' would be '$$2', or the result of adding '$$1' and '$$3'. By default, the value of a rule is the value of the first element on the right hand side, i.e $1 . All values are assumed to be integer.

The last section contains ratfor subroutines and functions. This section is optional. It will probably contain such user-supplied routines as the lexical analyzer, yylex, and the main program which will drive the generated parser,
yyparse.

FILES
none

SEE ALSO
"Yacc: Yet Another Compiler-Compiler" by S. C. Johnson "LR - Automatic Parser Generator and LR(tool) Parser" by C. Wetherell and A. Shannon.
yc(tutorial), yacc(tool), lrgen(tool), yyplb(lib)

AUTHOR
Theresa Breckon
NAME
acopy - copy n characters from file1 to file2

SYNOPSIS
call acopy( fd1, fd2, n )

integer fd1, fd2, n

DESCRIPTION
*Acopy* copies n characters from one open file to another. Neither file is repositioned before or after the copy. If an EOF is encountered on file1 the copy is terminated.

*Int1* and *int2* are internal file identifiers returned by calls to open or create.

*Acopy* simply makes repeated calls to getch and putch.

SEE ALSO
fsize(lib), fskip(lib), fcopy(lib), getch(lib), putch(lib)

DIAGNOSTICS
None
NAME
  addset - put character into array if it fits

SYNOPSIS
  integer addset, stat integer j, maxsize
  character c, array(maxsize)

  stat = addset( c, array, j, maxsize )

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.(as specified by
  'maxsize')

  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(lib)

DIAGNOSTICS
  None
NAME
addstr - add string s to str(j) if it fits, increment j

SYNOPSIS
YES/NO = addstr( s, str, j, maxsize )

integer addstr, j, maxsize character s(), str(maxsize)

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 copy­
ing is done and the status NO is returned.

Both s and str are ascii character arrays stored one character per array ele­
ment.

YES is returned if the routine succeeded, otherwise NO.

SEE ALSO
scopy, stcopy, addset, concat

DIAGNOSTICS
None

AUTHOR
STUG basic tape
NAME
alldig - return YES if string is all digits

SYNOPSIS
YES/NO = alldig( str )

integer alldig character str()

DESCRIPTION
Tests a string to see if it contains only digits. The string is an ascii character array terminated with an EOS marker.

Returns YES if string is all digits, otherwise NO.

SEE ALSO
type(lib)

DIAGNOSTICS
None
NAME
Amatch - look for pattern matching regular expression

SYNOPSIS
nexval = amatch( line, from, pat )

integer nexval, amatch, from character line(), pat(MAXPAT)

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 tutorial on the editor.)

Amatch is a special-purpose version of match, which should be used in most cases.

SEE ALSO
match(lib), getpat(lib), makpat(lib), ed(tool)

DIAGNOSTICS
A value of 0 is returned if the pattern does not match.

AUTHOR
Whei-Ling Chang

BUGS/DEFICIENCIES
NAME
Amatch - look for pattern matching regular expression

SYNOPSIS
nexval = amatch( line, from, pat )

integer nexval, amatch, from character line(), pat(MAXPAT)

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 tutorial on the editor.)

Amatch is a special-purpose version of match, which should be used in most cases.

SEE ALSO
match(lib), getpat(lib), makpat(lib), ed(tool)

DIAGNOSTICS
A value of 0 is returned if the pattern does not match.

AUTHOR
Whei-Ling Chang

BUGS/DEFICIENCIES
NAME

amove - move (rename) file1 to file2

SYNOPSIS

OK/ERR = amove( name1, name2 )

integer amove character name1(), name2()

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 ascii character arrays terminated with an EOS marker. If amove can not rename the file, it will return ERR status.

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. Or, if possible, it might be implemented by asking the system to rename the file.

SEE ALSO

fcopy,(lib) getlin(lib), putlin(lib)
NAME
appchr - append character onto the end of a string

SYNOPSIS
call appchr( char, str )

caller char, str()

DESCRIPTION
Appchr appends the character 'char' onto the end of the string 'str'. 'Str'
must be terminated by an EOS character.

SEE ALSO

DIAGNOSTICS

BUGS/DEFICIENCIES
NAME
appstr - append one string to another

SYNOPSIS
call appstr( temp, str )

character temp(), str()

DESCRIPTION
Appstr appends the string temp to the end of string str; an EOS character is then appended at the end of str. Both temp and str are character strings, so each must contain an EOS character before the call to appstr.

SEE ALSO

DIAGNOSTICS

BUGS/DEFICIENCIES
NAME
bslb - bit-string package

SYNOPSIS
include "bsdef"
call bsand( bsp1, bsp2, bsp3 ) bitarrays_are_equal = bsareq( bsvec1, bsvec2, numints ) call bsbcir( bsp, bitnum ) call bsbsset( bsp, bitnum ) bit_was_set = bsbtcn( bsp, bitnum ) bit_is_set = bsbtst( bsp, bitnum ) call bscomp( bsp ) call bscopy( bsp1, bsp2 ) call bsdestroy( bsp ) bitstrings_is_empty = bsempy( bsp ) call bsendloop( bsinx ) ) bitstrings_are_equal = bsequal( bsp1, bsp2 ) call bsgetbs( bsp1, bsvec1, numints ) call bsgetword( bsp, bwnum, bitword ) call bsinit( bsp, numbits ) call bsbinit bsinx = bsloopinit( bsp ) call bsbppos( bsinx, bitnum ) call bsmkcopy( bsp1, bsp2 ) another_element_found = bsnext( bsinx, val ) numbits = bsnmbits( bsp ) numints = bsnints( bsp ) call bsor( bsp1, bsp2, bsp3 ) call bsputbs( bsp1, bsvec1, numints ) call bsputword( bsp, bwnum, bitword ) call bsresetbit( bitword, bit ) call bssetbit( bitword, bit ) call bsstats bit_set = bstestbit( bitword, bit ) call bsxor( bsp1, bsp2, bsp3 ) call bszero( bsp ) BS_BITNUM_DECMP( bitnum, bitword, bit )

integer bsp, bsp1, bsp2, bsp3, bitnum, bsinx, numints, bwnum integer bitword, numbits, val, bit integer bsvec1(MAX_BS_SIZE), bsvec2(MAX_BS_SIZE)

integer bsloopinit, bsnmbits, bsnints logical bsareq, bsbtnc, bsbtst, bsempy, bsequal, bsnext logical bstestbit

DESCRIPTION
These routines implement a relatively fast set package based on bit strings. Bit strings are sets in which potential set elements have associated with them a non-negative integer. If the bit corresponding to a given potential set element is set in a bit string, then the potential set element is considered to be present in the set. If the bit is reset, then the potential set element is considered to not be in the set.

To use this package, the routine bslbinit must be called. Two types of bit strings may be used, and they are referred to in this document as bitstrings and bitarrays. Bitstrings are dynamic objects created by calls to bsinit by specifying how large a bitstring is desired (in bits). A "bitstring pointer" (an integer type) is returned which can be passed to other routines in the package to do various operations on the bitstring. Bitarrays are (presumably) static objects which are declared as integer arrays. The array can be passed to various routines to have bit operations done on its contents. Conversion between these two representations is facilitated by the bsgetbs and bsputbs routines. Bitarrays are not fully supported, and should only be used when fast, simple operations are desired.
The include file `bsdef` declares the macro `BS_BITNUM_DECOMP` (see below) for use with bitarrays. The file also contains three defines of user-interest: `BITS_IN_INTEGER` is the number of bits which can be stored in an integer value. `MAX_BS_SIZE` is the largest size (in integer words) that a bitstring can be (there is no limit on the size of bitarrays). Arrays used in calls to `bsgetbs` and `bsputbs` should be declared to be this size. Finally, `BS_MAX_LOOP_NESTING` is the greatest level to which bitstring loops can be nested (see `bsloopinit` below). The file need not be included if these defines are not needed.

The routines and their functions:

- `bsand` Produces the bit-wise "and" (intersection) of bitstrings `bsp1` and `bsp2` in bitstring `bsp3`. All bitstrings must have been previously initialized, must be the same size, and can all be the same bitstring.

- `bsareq` Returns true if two bitarrays, whose size in integer words is given by the `numirits` parameter, are equal, false otherwise.

- `bsbclr` Clears a bit in a bitstring.

- `bsbset` Sets a bit in a bitstring.

- `bsbtnc` Returns true if a specified bit is set in a bitstring, false otherwise. In any case, the bit is reset.

- `bsbtst` Returns true if a specified bit is set in a bitstring, false otherwise.

- `bscomp` Complements all the bits in a bit string (set negation).

- `bscopy` Copies bitstring `bsp1` into bitstring `bsp2`, which must have already been initialized. The bitstrings must contain the same number of bits.

- `bsdestroy` Deallocates the storage for a bitstring.

- `bempty` Returns true if a bitstring is empty, false otherwise.

- `bsendloop` Terminates a sorted search through a bitstring. Used in conjunction with `bsloopinit` and `bsnext`.

- `bsequal` Returns true if two bitstrings are equal, false otherwise.

- `bsgetbs` Returns the bitarray representation of a given bitstring, and the number of integer words the bitarray is in size. Companion routine with `bsputbs`. 
bsgetword Returns the specified bitword from a given bitstring. Used in conjunction with BS_BITNUM_DECOMP. Companion routine with bsputword.

bsinit Creates a bitstring large enough to hold the specified number of bits. A pointer to the bitstring is returned in bsp. The bitstring is initially empty (all bits reset). Enough room is allocated for bits numbered from zero to numbets (thus, for example, if "numbets" is 10, space is allocated for bits 0 thru 10, and the bitstring set can hold 11 distinct elements).

bslbinit Initializes the bit string library. Must be called before any other routines.

bsloopinit Prepares for a sorted loop through a bitstring. An index is returned as the function value. This index is repeatedly passed to bsnext to return the bit positions of set bits in the bitstring, from the lowest numbered bit to the highest, starting with bit 0. Loops should be terminated by calls to bsendloop with the index as argument. Up to BS_MAX_LOOP_NESTING bitstring loops may be active at one time. The loops needn't be nested (i.e. calls to bsnext may be made with different indices interspersed), but optimal speed results when bitstring loops are nested.

bslppos Sets a bit position for the next call to bsnext to begin at. For example,

    call bslppos( setindx, 24 )

would set things up so that the next time "setindx" was passed in a call to bsnext, the routine would begin searching for a set bit starting with bit 24. If bit 24 was reset, it would then test bit 25, 26, ... If all subsequent bits were reset, bsnext would fail (return false). It would not search bits numbered lower than 24.

bsmckopy Creates a copy of a bitstring. bsp2 should not already be initialized.

bsnext Returns in val the position of the next bit set in the bitstring associated with bsindx by a previous call to bsloopinit. Returns true if there was another set bit, false otherwise. For example,

    for ( i = bsloopinit( dataset );
        bsnext( i, dataval ); )
    call printf( "%d@n", dataval )

will write out, in increasing order, the numbers of all the elements in dataset.
bsnumbits Returns the number of bits allocated in a bitstring.

bsnumints Returns the number of integer words allocated for storage of a bitstring.

bsor Produces the bit-wise "or" (union) of bitstrings bsp1 and bsp2 in bitstring bsp3. All bitstrings must have already been initialized, must be the same size, and they can all be physically the same bitstring.

bsputbs Assigns bitstring bsp1 to the corresponding bitarray bsvec1, which contains numints integer words. No error checking is done to see if the bitarray is too large. Companion routine with bsgetbs.

bsputword Assigns a given bitword in the specified bitstring to the integer passed. Used in conjunction with BS_BITNUM_DECOMP. Companion routine with bsgetword.

bsresetbit Resets a specified bit of a given bitword. Used in conjunction with BS_BITNUM_DECOMP.

bssetbit Sets a specified bit of a given bitword. Used in conjunction with BS_BITNUM_DECOMP.

bsstats Writes bitstring usage statistics to error output.

bstestbit Returns true if a specified bit of a given bitword is set, false otherwise. Used in conjunction with BS_BITNUM_DECOMP.

bsxor Produces the bit-wise "exclusive or" of bitstrings bsp1 and bsp2 in bitstring bsp3. All three bitstrings must have previously been initialized, must be the same size, and can be the same physical bitstring.

bszero Resets all the bits in a given bitstring.

BS_BITNUM_DECOMP is a macro which takes as an argument an integer representing a bit position in a bitarray. It assigns to its second argument the word number (starting at zero) in which that bit position would be found in the bitarray, and it assigns to its third argument the bit number corresponding to where the bit position would be found in the bit word. For example, if

BS_BITNUM_DECOMP( 37, bitword, bit )

assigned bitword = 2 and bit = 5, then the 37th bit position of any bitarray would be the 5th bit of the bitword #2 (remember that bitwords are numbered starting at zero).
SEE ALSO
  quelb(lib)

DIAGNOSTICS
  "error in memget", "error in memput", etc. usually means that a routine
  was passed a bitstring pointer which was not initialized.

  "loops nested too deeply" usually means that you forgot to call bsendloop at
  the termination of a bitstring search.

  "bit number out of range" means that an attempt was made to reference a
  bit position in a bitstring which was beyond the upper limit established when
  the bitstring was initialized by bsinit.

AUTHOR
  Vern Paxson

BUGS/DEFICIENCIES
  Bitstrings may have at most 5000 elements.

  Most operations (e.g. bit-wise "and") are available only for bitstrings, and not
  bitarrays.

  Little or no error-checking is done in the low-level routines.

  The library relies on intrinsic Fortran functions "iand", "not", "ishft", "ior",
  and "ieor", and thus is quite non-portable.
NAME
bubble - bubble sort an integer array (increasing order)

SYNOPSIS
call bubble( ray, n )
integer ray(), n

DESCRIPTION
Bubble sorts the first n elements of integer array 'ray' in increasing order.
This means that after calling bubble, ray(1) will contain the smallest value.
Ray is an integer array, and n is an integer.

SEE ALSO
shell(lib)

BUGS/DEFICIENCIES
There is no reason to have two separable sort routines which perform exactly
the same function. Since 'shell' is probably faster, it should be used instead
of 'bubble'. 'Bubble' is kept for compatibility.
NAME
cant - prints 'name: can't open' and aborts execution

SYNOPSIS
call cant( name )

character name()

DESCRIPTION
On ERROUT, prints the filename specified by 'name' followed by the message
"can't open" and aborts execution. All open files are closed. Name is an ascii
character array terminated with an EOS marker.

SEE ALSO
error(lib), remark(lib)

DIAGNOSTICS
None
NAME
catsub - add replacement text to the end of new buffer

SYNOPSIS
call catsub( lin, from, to, new, k, maxnew )

integer from, to, k, maxnew character lin(MAXLINE), new(maxnew), sub()

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'.

SEE ALSO
getpat(lib), makpat(lib), amatch(lib), getsub(lib), maksub(lib)

DIAGNOSTICS

AUTHOR
Whei-Ling Chang

BUGS/DEFICIENCIES
NAME
  `chcopy' - append character to string at specified character

SYNOPSIS
  call chcopy( char, out, j )

  character char, out() integer j

DESCRIPTION
  Chcopy copies character char to string 'out' at index j, and then appends an
  EOS character at index j + 1. It is not necessary for out to contain an EOS
  character before the call to chcopy. (Thus 'out' may not really be a real
  string) The index j gets incremented to point to the new end of the string.

SEE ALSO

DIAGNOSTICS

BUGS/DEFICIENCIES
NAME
chdir - change working directory

SYNOPSIS
OK/ERR = chdir(dir)

integer chdir character dir()

DESCRIPTION
Chdir changes the default working directory. 'Dir' is a character array terminated with a EOS character. This function returns either OK or ERR. OK indicates that the working directory has been changed; ERR indicates that the new directory does not exist or has invalid syntax.

BUGS/DEFICIENCIES
On the Modcomp, the status is returned as YES. An error message is printed on the terminal by the library if chdir didn't work.
NAME
chkdir - check to see if passed file is a directory

SYNOPSIS
YES/NO = chkdir(dir)

integer chkdir character dir()

DESCRIPTION
Chkdir is a function used to check to see if the name in the passed character
string 'dir' is a directory. Chkdir returns YES if 'dir' is a directory, NO if it is
not.

BUGS/DEFICIENCIES
Determining the difference between a file and a directory may sometimes be
machine-dependent, thus this routine should be used with caution. For
example, if on the Vax you have two files that are called, for example, "foo"
and "foo.dir", calling chkdir with "foo" will return YES even though the file
you were interested in might have been "foo" and not "foo.dir"
NAME
  close - close (detach) a file

SYNOPSIS
  call close( int )

    integer int

DESCRIPTION
  Close closes the connection between a file and the running program. Any
  write buffers are flushed and the file is rewound.

  "Int" is an internal file descriptor as returned from an open or create call.

  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(lib), create(lib)

DIAGNOSTICS
  If the file descriptor is in error, the routine simply returns.
NAME
   clower - fold passed a character to lower case

SYNOPSIS
   c = clower( c )

   character c, clower

DESCRIPTION
   Fold character 'c' to lower case, if not already. If 'c' is not alphabetic,
   returns it unchanged.

SEE ALSO
   fold(lib), cupper(lib), upper(lib)

DIAGNOSTICS
   None
NAME

cmatch - try to find character in string

SYNOPSIS

cchar/EOS = cmatch( char, ray )

character cmatch, char, ray()

DESCRIPTION

Cmatch scans the character array 'ray' for the character 'char'. If 'char' is in the string, then 'char' is returned, otherwise an EOS character is returned. 'Ray' must contain an EOS character.

SEE ALSO

DIAGNOSTICS

BUGS/DEFICIENCIES
NAME
concat - concatenate the two passed strings

SYNOPSIS
call concat(str1, str2, dest)

character str1(), str2(), dest()

DESCRIPTION
Concatenate str1 and str2 and put the result string into dest.

SEE ALSO
stcopy, scopy, addset, chcopy

DIAGNOSTICS
NAME
create - create a new file or overwrite if existing

SYNOPSIS
int/ERR = create( name, access )

integer create, int, access character name()

DESCRIPTION
Creates a new file from within a running program and connects the external
name of the file to an internal identifier. If the file already exists, the old
version will be overwritten.

"Name" is a character string representing a pathname or filename in whatever format is used by the local operating system. It may be passed as a quoted string or as an array of ascii characters terminated with an EOS marker.

"access" is a descriptor for the type of access desired - READ, WRITE, READWRITE, or APPEND.

"int" is returned as a one-word integer internal descriptor to the file.

Create is similar to open except that create generates a new file. If the file already exists, the old version is removed or truncated and overwritten. All other functions are similar to open.

If the file is new, the file type (ASCII or LOCAL) should be set to whichever default you have chosen.

SEE ALSO
open(lib), close(lib)

DIAGNOSTICS
The function returns ERR if the file could not be created or if there are already too many files open.
NAME
ctoc - copy string-to-string, observing length limits

SYNOPSIS
numchr = ctoc( from, to, len )

integer ctoc, numchr, len character from(), 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. 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 (lib), other conversion routines ('cto*' and '*toc') (lib)

AUTHOR
STUG basic tape
NAME
cstoi - convert string at specified location to integer

SYNOPSIS
n = cstoi( str, idx )

integer cstoi, n, idx character str()

DESCRIPTION
Converts a character string to an integer. Starts looking at position 'idx' of 'str'. Leading blanks and tabs are ignored. A single optional plus or minus sign is allowed and interpreted correctly; 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 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(lib)

DIAGNOSTICS
There are no checks for machine overflow.
NAME
ctomn - translate ASCII control character to mnemonic

SYNOPSIS
len = ctomn( c, rep )

integer ctomn, len character c, rep(4)

DESCRIPTION
'Ctomn' is used to convert an unprintable ASCII character to its official ASCII mnemonic. The first argument is the character to be converted; the second is a string to receive the mnemonic. The function return is the length of the string placed in the second argument.

If the character passed is printable, it is copied through unchanged to the receiving string. If not, its two- or three-character ASCII mnemonic (e.g. NUL, SOH, etc.) is copied into the receiving string. If the character is printable, it is placed in the receiving string, which is then terminated with EOS. If the character is between 0 and 32, inclusive, or equals 127, its value is used to compute an index into a string table containing the mnemonics. The mnemonic thus selected is copied into the receiving string.

SEE ALSO
mntoc (lib)

AUTHOR
STUG basic tape
NAME
ctor - convert character string to real number

SYNOPSIS
val = ctor( str, idx )

real ctor, val character str() integer idx

DESCRIPTION
CTOR takes a character string, 'str' and converts it to a real number, begin­ning at position 'idx'. 'Idx' is then incremented to point to the first character after the converted string. CTOR takes a real number as defined by the regu­lar expression:

\[ [+]-d*.[d*][(e|E)[+|-]d*] \]

and returns the real value, 'val'. Note that leading '+' and '-' signs are correctly interpreted, and 'E' notation is understood.

'Str' is an ascii character array terminated by an EOS marker.

Note that the type of ctor is real.

SEE ALSO
rtoc(lib)

FILES
None

AUTHOR
Van Jacobson

BUGS/DEFICIENCIES
NAME
cupper - convert character to upper case

SYNOPSIS
c = cupper(c)

character cupper, c

DESCRIPTION
Converts ascii character 'c' to upper case, if not already upper case. Non-alphabetic characters are returned unchanged.

SEE ALSO
upper(lib), clower(lib), fold(lib)

DIAGNOSTICS
None
NAME
cyplb - cypher library (encryption algorithms)

SYNOPSIS
call encrypt( line, key, keyptr ) c = xor( a, b )

logical xor, c character line(), key(), a, b integer keyptr

DESCRIPTION
Cyplb is a collection of routines used to encrypt and decrypt character strings. The following routines exist in this library:

- **Encrypt** is a subroutine which encrypts a string of characters according to the passed key. 'Line', the first parameter, is a ratfor string containing the characters to be encrypted. 'Key', the second parameter, is another ratfor string containing the key to be used. 'Keyptr' is an integer which gives the offset into key of the next character to be used in the encryption process. Upon return, 'line' will return with the encrypted line replacing the passed cleartext, and keyptr will return with the value of the offset into key of the next character to be used in the next call to encrypt. (The use of keyptr is to make encrypt compatible with older versions of the crypt tool.)

- **xor** is a function which will return the logical exclusive or of two character values. The two parameters, 'a' and 'b', are passed as characters, and the result of this function (also a character) is the exclusive or of 'a' and 'b'.

FILES

SEE ALSO
NAME

delarg - delete command line argument number 'n'

SYNOPSIS

call delarg( n )

integer n

DESCRIPTION

Delarg deletes reference to the command line argument number 'n' so that subsequent calls to 'getarg' do not see it.

Delarg works in conjunction with 'getarg'. It generally re-orders indices to an array holding the command line arguments. If argument 'n' does not exist, delarg simply returns.

SEE ALSO

getarg(lib), initst(lib)
NAME
dospwn - spawns a shell command

SYNOPSIS
OK/ERR/CHILD_ERROR = dospw\n( str )

integer dospw\n character str()

DESCRIPTION
Dospwn executes the string 'str' as a shell command. 'Str' is a character array. If 'str' is null (i.e. str(1) = EOS) then an interactive shell is spawned. The shell that is used is located via the shell search path. 'Str' may contain any characters which can normally be handled by an interactive shell. In other words, pipes, file redirection, etc., work as expected. Return status is OK, ERR, or CHILD_ERROR.

SEE ALSO
spawn(lib)
NAME
dslb - ratfor dynamic memory allocation routines

SYNOPSIS

```
DS_DECL[( MEMSIZE )] status = dsaget( index, array, count, position )
status = dsaput( index, array, count, position )
status = dscop( index1, offset1, index2, offset2, count )
status = dseget( index, element, position )
status = dsfree( index )
status = dsget( index, count, type )
```

```
integer dsget, dsfree, dsaput, dsaget, dscop, dseput, dseget integer count, index, type, position
```

DESCRIPTION

Dslb is a collection of routines that allow manipulation of dynamically allocated memory. The dynamic memory is accessed through a machine-dependent pointer, 'index'. This pointer has type and word size equivalent to integer. The initialization macro, DS-PECL, must be specified before the call to dsget. The optional argument MEMSIZE is the total size of the dynamic memory area. If MEMSIZE is not specified, a default value is used. The routines are:

**Dsaget** retrieves 'count' elements from the dynamic memory space referenced by 'index' and stores these elements into the array 'array'. 'Array' is assumed to be able to handle 'count' number of elements of the type specified in the initial 'dsget' call for 'index'.

The elements are retrieved from the dynamic memory area starting with the 'position'th element. If an attempt is made to retrieve more elements than exist in the dynamic memory area, dsaget only copies as many elements as exist.

If 'position' is not within the range of the dynamic memory area, or if 'index' is INSUFFICIENTMEMORY, ERR is returned, otherwise OK is. Note that the contents of the memory area will be unpredictable unless information is put in it first with dsaput or dseput.

**Dsaput** stores 'count' elements from the array 'array' into the dynamically allocated memory area 'index'. 'Array' is assumed to carry at least 'count' number of elements of the same type as specified in the original 'dsget' call for 'index'.

The elements are stored into the dynamic memory area starting at the 'position'th element. If an attempt is made to store elements past the end of the dynamic memory area, dsaput will stop storing elements at the end of memory area.

If 'position' is not within the range of the dynamic memory area, or if 'index'
is INSUFFICIENTMEMORY, ERR is returned. Otherwise OK is returned.

**Dscop** copies a segment of memory from one dynamic area to another dynamic area.

'Index1' is the address of the first dynamic memory segment, and 'ofset1' is its displacement. The destination of the copy is represented by 'index2' and its displacement by 'ofset2'. 'Count' corresponds to the number of elements to be moved to the destination area. It is not an error if the two dynamic memory areas are not of the same type, however it is highly recommended that they are the same type.

If 'index1' or 'index2' is INSUFFICIENTMEMORY, or if either 'ofset1' or 'ofset2' is not within the range of the dynamic memory, ERR is returned. Otherwise OK is returned.

**Dseget** retrieves an element from the dynamic memory space referenced by 'index' and stores it into the variable 'element'. 'Element' is assumed to be of the same type as the dynamic memory space.

The data is retrieved from the 'position'th element of the dynamic memory area.

If 'position' is not within the range of the dynamic memory area, or if 'index' is INSUFFICIENTMEMORY, ERR is returned, otherwise OK is returned. Note that the contents of the memory area will be unpredictable unless information is put in it first with dsaput or dseput.

**Dseput** stores an element into the dynamically allocated memory area 'index'. 'Element' is assumed to be of the same type as the memory area, as specified in the original 'dsget' call for 'index'.

The element is stored into the dynamic memory area at the 'position'th element.

If 'position' is not within the range of the dynamic memory area, or if 'index' is INSUFFICIENTMEMORY, ERR is returned. Otherwise OK is returned.

**Dsfree** frees memory allocated dynamically by dsget.

'Index' is the index of the memory area, returned by dsget. If 'index' was set to INSUFFICIENTMEMORY by dsget, ERR is returned, otherwise OK is returned.

Dsget allocates memory dynamically for a block of 'count' number of elements of type 'type'. Dsget will return ERR if either 'type' or 'count' are equal to zero. For example, to allocate memory for 40 integers the call to dsget would be
sts = dsget( index, 40, DS_INTEGER )

Other valid 'types' are DS_CHAR, DS_LOGICAL, and DS_REAL. 'Index' is used by the other dynamic memory allocation routines to reference the memory space. If not enough memory is available, index is set to INSUFFICIENTMEMORY and ERR is returned, otherwise OK is returned.

FILES
/usr/incl/ratdef: contains definition for dynamic memory (DS_CHAR, etc.)

SEE ALSO
tblib(lib)

AUTHOR
Theresa Breckon
NAME
endst - do system-dependent cleanup and terminate tool

SYNOPSIS
 call endst( status )

 integer status

DESCRIPTION
Endst is normally called at the end of any program which uses the software tools primitives. It closes all open files and terminates program execution.

On many systems a call to endst is made automatically, either by the system or by the DRIVER macro in the standard definitions file.

The program's termination "status" should be passed as OK or ERR. A status of ERR will result in the 'parent' of this program receiving a CHILD_ERROR status return from its call to spawn. Any open files are closed. If any files have been opened multiple time (that is, they have more than one internal descriptor assigned to them), care is taken that multiple closes do not damage the file.

SEE ALSO
close (lib), initst (lib), spawn (lib)

DIAGNOSTICS
 none
NAME
envlb - portable environment manipulation functions

SYNOPSIS
OK/ERR = envset( name, value )
OK/ERR = envget( name, value )
OK/ERR = envrmv( name )
OK/ERR = envprt( fd )

integer envset, envget, envrmv, envprt, fd
character name(), value()

DESCRIPTION
These functions are used to manipulate environment variables. "Name" is a
character array containing the name of the environment variable. "Value" is
a character array representing the value of the environment variable. Vari‐
able names are case insensitive, i.e. the environment routines treat "PATH"
and "path" as the same environment variable. Both character strings are
assumed to be terminated with EOS.

Envset - assigns "value" as the value of the environment variable identified
by "name". Any tool spawned following the "envset" call will inherit "name"
and its "value". If the value could not be set for the variable, ERR is
returned, otherwise OK is.

Envget - retrieves the value for an environment variable. The value will be
returned in the character array "value". If the environment variable doesn't
exist, ERR is returned, otherwise OK is returned.

Envrmv - removes an environment variable from the set of defined vari‐
ables. If the variable could not be removed, ERR is returned, otherwise OK is.

Envprt - outputs the users environment onto the file descriptor "fd". The
value of each environment variable is output as "name=value".

AUTHOR
Theresa Breckon

SEE ALSO
setenv(tool), unsetenv(tool), resetenv(tool), printenv(tool), sh(tool)
NAME
equal - compare two strings

SYNOPSIS
YES/NO = equal( str1, str2 )

integer equal character str1(), str2()

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

DIAGNOSTICS
None
NAME
   error - outputs one-line error message and aborts execution

SYNOPSIS
   call error( line )
   character line()

DESCRIPTION
   Outputs the error message line to ERROUT and aborts further execution.
   The message array is generally a Ratfor double quoted string. For example,
   
   call error( "gronk: symbol table overflow" )

   Line may also be an ascii character array terminated by an EOS marker.

FILES
   None

SEE ALSO
   remark(lib), putlin(lib)

AUTHOR

DIAGNOSTICS
   None

BUGS/DEFICIENCIES
NAME
esc - map element of array(i) into escaped character

SYNOPSIS
\[
c = \text{esc}( \text{array, i})
\]
character array(), esc, c integer i

DESCRIPTION
Esc checks 'array(i)' for an escape character (the character defined by
ESCAPE in the general symbol definitions). If an escape is found, the escape
sequence starting at 'array(i+1)' is interpreted as described below and 'i' is
incremented to point to the last character of the escaped sequence. If no
escape is found, the character 'array(i)' is returned and 'i' is not incre­
mented.

Possible escapes sequences 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
^c  control characters ('c' is any upper or lower
case alphabetic character or one of the
following: @,[,],^,? . For example,
^G is turned into an Ascii 'BEL' character).
000 (‘zero’ followed by exactly three octal
digits, the largest octal value possible
being 177) is interpreted as the character
with that octal value; e.g., 021 would be
an Ascii ‘DC1’. If an invalid octal value (over 177)
is given the character ‘0’ will be returned.

If the sequence following the escape character does not confirm to any of the
specifications listed above, the character following the escape is returned as
the function and i is incremented to point to this character, e.g. “\@02b”
would return the character zero, and i would be incremented by one.

SEE ALSO
index(lib), type(lib)
DIAGNOSTICS
NAME

evalb - expression evaluation routine library

SYNOPSIS

true./false. = bsep(ptr)
true./false. = evalexpr(exp, result)
true./false. = lprnp(ptr)
call cpush(opr, ptr)
call putreal(number, min-wid)
call rpush(r4, ptr)
value = rev(arg1, opr, arg2)

logical bsep, evalexpr, lprnp
integer opr, ptr, min-wid
character exp()
real result, number, rev, value, arg1, arg2, r4
integer*4 opr

DESCRIPTION

Evalb is the standard software tools expression evaluation library. The routines
it contains are used by such tools as macro, roff, rofmac, etc.

In the following section, each routine is described.

bsep - whether 'ptr' is at the beginning of a
sub-expression or not if ptr is zero, or if it points to a
left paren, then bsep returns true.; otherwise, it is
false.
passed
ptr - integer, assumed to be a pointer into
cstk or typstk.
returned
status - true. or false.

cpush - push an operator onto the stack
pushes the operator 'opr' onto the stack 'cstk' at
'ptr', and the value OP onto the stack 'typstk' at
'ptr'.
passed
opr - the operator to be pushed onto 'cstk'
ptr - integer, assumed to be a pointer into 'cstk'
    and 'typstk'

evalexpr - evaluate an arithmetic expression
in a string evaluates the arithmetic expression in the
string 'exp' and returns the resulting value
in the real 'result'. If all went well, evalexpr
itself returns true.; if not, it returns false.
and 'result' is not set. Valid operators are PLUS, MINUS, STAR, SLASH, and CARET.

Passed
exp - the string to be evaluated as an arithmetic expression

Returned
result - the value of 'exp' (real)
status - .true. or .false.

lprnp - whether ptr points to a left paren or not
if typstk(ptr) is not equal to OP, or if cstk(ptr) is not a left paren, lprnp is .false.; otherwise, it is .true.

Passed
ptr - integer, assumed to be a pointer into typstk and cstk.

Returned
status - .true. or .false.

putreal - output a real number
outputs the real number 'number', in a field of minimum width 'minvid', onto SDTOUT.

Passed
number - real number to be output
minvid - the minimum size of the field in which 'number' is to be output

rev - evaluate real binomial
evaluates the expression arg1 opr arg2

Passed
arg1, arg2 - real
opr - character, representing the operation to be performed

Returned
value - real

rpush - push a real onto the stack.
pushes the real 'r4' onto the stack 'rstk' at 'ptr', and the value RL onto the stack 'typstk' at 'ptr'.

Passed
r4 - real, to be pushed onto the stack
ptr - integer, assumed to be a pointer into rstk and typstk
SEE ALSO
   roff(tool), macro(tool), rofmac(tool)
NAME
   fcopy - copy one file to another

SYNOPSIS
   call fcopy( in, out )
   
   integer in, out

DESCRIPTION
   Fcopy assumes 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.

SEE ALSO
   acopy(lib)

DIAGNOSTICS
   None
NAME
    flush - flush output buffer for file 'fd'

SYNOPSIS
    call flush( fd )

        integer fd

DESCRIPTION
    Flush assures that any remaining characters in the output buffer of the file
    specified by "fd" are sent out to the file. It is useful for sending lines to a
    teletype-like device without requiring a NEWLINE character, and also for
    flushing buffers after calls to "writef".

IMPLEMENTATION
    It is expected that most software tools installations will employ some form of
    buffered I/O. Flush is intended to define the buffer-clearing operation that
    takes place before file closing, and to provide a means of insuring that output
    directed to a terminal has appeared on that terminal (e.g. before obtaining
    some input after a prompt). On systems with unbuffered I/O, flush is a no-op.

SEE ALSO
    prompt, writef, putch, putlin

DIAGNOSTICS
    None
NAME
fmtdat - convert date information to character string

SYNOPSIS
call fmtdat( date, time, now, form )

character date(), time() 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; at present, the following forms are allowed:

STANDARD  DDMmmyY HH:MM:SS
SORTABLE  YYYY-MM-DD HH-MM-SS

SEE ALSO
getnow(lib), date(tool)

AUTHOR
Marshall Spight

BUGS/DEFICIENCIES
The current form returned when STANDARD is selected is the form used by RTSG, and not the STUG STANDARD.
NAME
fnlb - library routines to deal with file names

SYNOPSIS
call addext( fname, ext ) call delext( fname ) YES/NO = extext( fname, ext )
ind = getext( fname ) ptr = lastsn( fname )

# the following are implemented on the VAX only: call addver( fname, version )
call delver( fname ) YES/NO = extver( fname, version ) ind = getver( fname )

integer ind, ptr, version character*fname(), ext()

DESCRIPTION
fnlb is the software tools file name manipulation library.

On the VAX, all routines will allow the user to specify file names in either
tools or files-II format. Also note that for the VAX, "x.y.z" is equivalent to
"x.y;z". Each routine is described in the following section.

1. Routines Available on all Computers

 addext - Add extension to file name. Any previous extension is deleted.
Any previous version number is preserved. 'Fname' contains the file name to
which the extension is added. 'Ext' is a string which contains the extension
to be added. Here are some examples:

call addext( "x", "a" ) --> "x.a"
call addext( "x.", "a" ) --> "x.a"
call addext( "x.y", "a" ) --> "x.a"
call addext( "x..z", "a" ) --> "x.a.z" # VAX only
call addext( "x.y.z", "a" ) --> "x.a.z" # VAX only

delext - Deletes extension from file name. If no valid extension exists,
nothing is done. Any version number present is preserved. 'Fname' contains
the file name from which the extension is to be deleted. Here are some exam­
pies:

call delext( "x" ) --> "x"
call delext( "x.y" ) --> "x" # other than VAX
call delext( "x." ) --> "x." # VAX only
call delext( "x..z" ) --> "x..z" # VAX only
call delext( "x.y" ) --> "x." # VAX only
call delext( "x.y.z" ) --> "x..z" # VAX only

extext - Extract file name's extension from the file name contained in the
array 'fname'. 'Fname' is not changed. YES is returned only if a valid exten­sion was found. 'Ext' is set to a null string if NO extension was found, other­wise 'ext' will contain the extracted extension string. The extension
character is NOT returned as part of the extension. Here are some examples:

```
extest("x")  --> NO, ""
nextest("x.y") --> YES, "y"
nextest("x.")  --> NO, ""  # VAX only
nextest("x.y.") --> YES, "y"  # VAX only
nextest("x.z") --> NO, ""  # VAX only
nextest("x.y.z") --> YES, "y"  # VAX only
```

g

gnextest - Returns a pointer to the beginning of the extension of the file name contained in 'fname'. Note that '..' and '.<end of string>' are not valid extensions on the VAX. NOT_FOUND is returned if the file name in 'fname' has no valid extension. Here are some examples:

```
gnextest("123")  --> NOT_FOUND
gnextest("123.56") --> 4
gnextest("123.")  --> NOT_FOUND  # VAX only
gnextest("123.67") --> NOT_FOUND # VAX only
gnextest("123.56;" ) --> 4  # VAX only
gnextest("123.56;89") --> 4  # VAX only
```


g

glastsnn - Returns a pointer to the last simple name of the file name contained in 'fname'. The pointer will point to the first character of the last simple name in 'fname'. Here are some examples:

```
glastsn("xxx/yyy")   --> 5
glastsn("/xxx/yyy/zzz") --> 10
glastsn("xxx.yyy;zzz") --> 1  # VAX only
glastsn("[xxx]yyy") --> 7  # VAX only
glastsn("dv:[xxx.yyy][zzz]") --> 13 # VAX only
glastsn("xxx:yyy")  --> 5# VAX only
```

2. Routines for VAX Only

addver - Adds a version number to the file name contained in 'fname'. Any previous version number is replaced. The version number is passed as an integer value in 'version'. Here are some examples:

```
call addver( "x.y;z", "a" ) --> "x.y:a"
call addver( "x.y;" , "a" ) --> "x.y:a"
call addver( "x.y", "a" )  --> "x.y:a"
call addver( "x.z", "a" )  --> "x.a"
call addver( "x", "a" )   --> "x.a"
call addver( "x", "a" )   --> "x.a"
```

d

delver - Delete version number of file name. Here are some examples:

```
call delver( "x.y.z" )  --> "x.y"
call delver( "x.z" )    --> "x."
```
call delver( "x.y" ) --> "x.y"
call delver( "x." ) --> "x."
call delver( "x" ) --> "x."

**extver** - Extract version number from file name. A version number of 0 is returned if no version number was present in 'fname'. The integer version number is returned in 'version'. The function value is YES if there was a version number, NO if not. Here are some examples:

- extver("x.y.2") --> YES, 2
- extver("x..3") --> YES, 3
- extver("x.y.") --> YES, 0
- extver("x.y") --> NO, 0
- extver("x..") --> NO, 0
- extver("x") --> NO, 0

**getver** - Returns a pointer to the delimiter character of the version number in the file name contained in 'fname'. This routine differs in structure from getext in that "12.45;" is considered to have a valid version number starting at position 6. The function value is an integer; the pointer to the character position of the version number character delimiter. NOT_FOUND is returned if there is no version number. Here are some examples:

- getver("123.56;89") --> 7
- getver("123.56;") --> 7
- getver("123.56") --> NOT_FOUND
- getver("123..67") --> 5
- getver("123..") --> 5
- getver("12.") --> NOT_FOUND
- getver("12") --> NOT_FOUND

**BUGS/DEFICIENCIES**

File names ending with a slash ("/"") will generate unpredictable results that are very likely not what the user expected.
NAME
  fntopn - convert file name to path name

SYNOPSIS
  OK/ERR = fntopn( fname, pname )

  integer fntopn character fname(), pname()

DESCRIPTION
  Fntopn converts the file name to an unambiguous full path name. Both
  'fname' and 'pname' are ratfor strings.

  If 'fname' is a legal file name that fntopn can convert to a full path name,
  fntopn returns OK. Otherwise, fntopn returns ERR.

  On some machines, 'fname' must only be in tools file name format. Others
  allow 'fname' to also be in a machine-dependent format.

  On the VAX, 'fname' may be in Files-11 format.

SEE ALSO
  paths(info)
NAME
fold - convert string to lower case

SYNOPSIS
call fold(str)

character str()

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(lib), cupper(lib), upper(lib)

DIAGNOSTICS
None
NAME
fprintf - formatted output to a file

SYNOPSIS
 call fprintf(fd, fmt, arg1, arg2, ...)

integer fd character fmt()

DESCRIPTION
fprintf() converts, formats, and prints its arguments under control of a for­
matter specification. It allows the user to specify a file to write to, a format to
control the output, and a number of items to be printed. In fprintf(), "fd" is
the file descriptor of the file to be used for output, "fmt" is a format string
(discussed below), and the remaining arguments (zero or more) are items to
be output according to format control.

The options available in a format string are discussed in the manual entry for
prints(lib).

EXAMPLES
To write a string, followed by a new line ("@n") onto the file specified by "fd":
   call fprintf(fd, "%s@n", string)

To write two real numbers, the first in a field of five (5) with a precision of two
(2), and the second in a standard format, along with some text for
identification, followed by a newline:
   call fprintf(fd, "x = %f5.2, y = %f@n", xcoord, ycoord)

To convert an integer to characters:
   call fprintf(fd, "%d", int)

SEE ALSO
prints(lib), printf(lib)

AUTHOR
Rick Corona

BUGS/DEFICIENCIES
The format string may be a ratfor string or, on some systems, may also be a
quoted or hollerith string. For example, use of quoted or hollerith strings will
work on a Vax, but not on the Modcomp. It is recommended that hollerith
strings not be used in programs if they are to be truly portable.

There can be no whitespace between the routine name and the left
parenthesis of the parameter list. This is due to the fact that macro pre-
processing does not allow spaces between the macro name and the definition following it.

There is a limit to the number of arguments that can be formatted, depending on the machine being used. The current RTSG machines limit is seven (7) arguments.
NAME
fsize - determine size of a file (in characters)

SYNOPSIS
size = fsize( filename )

integer fsize, size character filename()

DESCRIPTION
Fsize opens the file specified by 'filename', and determines its size in characters. It then closes the file. NEWLINES are counted as characters.

The filename is passed as an ascii character array terminated with an EOS marker.

The normal implementation of fsize is to have it count calls to 'getch' before an EOF is encountered. However, some systems provide more efficient ways of determining a file's size.

This routine is generally used by the archiver to store a file's size in the archive header.

SEE ALSO
acopy(lib), fskip(lib), getch(lib)

DIAGNOSTICS
ERR is returned if the file could not be opened.
NAME
  fskip - skip n characters on a file

SYNOPSIS
  call fskip( int, n )

integer int, n

DESCRIPTION
  From the current position in the file specified by 'int', fskip skips 'n' characters forward. NEWLINES are counted as characters.

  The most common implementation of fskip is to have it call getch for the appropriate number of times. However, in more congenial systems it may be possible to have the routine skip without reading the intervening data.

  Fskip is generally used by the achiver to skip over its internal files.

SEE ALSO
  acopy(lib), fsize(lib), getch(lib)

DIAGNOSTICS
  None
NAME
gctoi - generalized character-to-integer conversion

SYNOPSIS
convint = gctoi( str, i, radix )

integer convint, gctoi, radix, i character str()

'Gctoi' is similar to the routine 'ctoi', except that it accepts base indicators and signs. Conversion begins on the string 'str' at position 'i'. The converted integer is returned as the value of the function. 'i' will be updated to indicate the first position not used in the conversion.

Input to 'gctoi' consists of a number containing an optional plus or minus sign, an optional base indicator, and a string of digits allowable for the input base. The base indicator consists of the (decimal) radix of the desired base followed by the letter "r" (in the style of Algol 68). The digits corresponding to the numbers 10 through 15 are entered as the letters "a" through "f". If no base indicator occurs in the string, the number in 'radix' is used as the default base. Conversion stops when a character not allowable in the number is encountered. 'Gctoi' first checks for a leading sign, and records it if found. If the first one or two digits of the number are numeric and if they are followed by a lower case "r", then they are converted to binary and used as the radix of the remaining digits; otherwise, the 'radix' argument is used. The remaining digits of the number are converted by a simple multiply-and-add-successive-digits algorithm.

ARGUMENTS MODIFIED

i

CALLS

index

SEE ALSO

ctoi (lib), other conversion routines ('cto?' and '?toc') (lib)

AUTHOR

STUG basic tape
NAME
gdate - get current date

SYNOPSIS
call gdate( date )

character date()

DESCRIPTION
'Gdate' is used to get the current date, which is returned into 'date', a character string, in the format:

ddMmmyy

SEE ALSO
gtime(lib), date(tool), ar(tool)

BUGS/DEFICIENCIES
Only the last two digits of the year are returned, which will cause problems in the year 2000.
NAME
getarg - get command line arguments

SYNOPSIS
len/EOF = getarg( n, array, maxsize )

integer getarg, len, n, maxsize character array(maxsize)

DESCRIPTION
'Getarg' gets command arguments from the command line or control card
and copies the 'n' th command line argument into the ASCII character array
'array', terminating it with an EOS marker. 'Maxsize' is passed as the max­
imum number of characters array is prepared to deal with (including the
EOS marker); getarg truncates the argument if necessary to fit into the
space provided. The number or characters in the argument (not including
the EOS marker) is returned in the function call. If there are less than 'n'
arguments, EOF is returned.

The implementation of 'getarg' may be quite different on different operating
systems. Some systems allow only upper case (or lower case) on the com­
mand 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 argu­
ments.

If it is absolutely impossible to pick up command line arguments from the
system, 'makarg' could be taught to prompt the user for them.

When the shell is implemented, 'getarg' (or perhaps 'makarg') will have to be
altered to read arguments as passed from the shell.

SEE ALSO
initr4(lib), delarg(lib)

DIAGNOSTICS
None
NAME
  getc - get character from STDIN

SYNOPSIS
  char = getc( char )

  character getc, char

DESCRIPTION
  Getc reads the next character from STDIN. The character 'char' is returned
  both as the functional return value of the function and in the parameter
  'char'. If the end of a line has been encountered, a NEWLINE is returned. If
  the end of file has been encountered, EOF is returned.

SEE ALSO
  getch(lib)
NAME
getch - read character from file

SYNOPSIS

\[ c = \text{getch}(\text{c}, \text{int}) \]

c character getch, c integer int

DESCRIPTION

Getch reads the next character from the file specified by int. 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.

Interspersed calls to getch and getlin work properly. A common implementation is to have getlin work by repeated calls to getch.

Getch is able to recognize an end-of-file marker from either a terminal or a file.

SEE ALSO
getlin(lib), putch(lib), putlin(lib), getc(lib)

DIAGNOSTICS
None
NAME
getlin - get next line from file

SYNOPSIS
len/EOF= getlin( line, fd )

integer getlin, fd character line(MAXLINE)

DESCRIPTION
Getlin copies the next line from the file with the internal name fd into the
character array line. Characters are copied until a NEWLINE marker (end­
of-record marker, or whatever method is used to determine an end-of-line by
the local operating system) is found or until MAXLINE characters have been
copied. A NEWLINE character is returned whenever an end-of-line marker
has been sensed. The characters are returned in an ascii character array
terminated with an EOS marker (possibly preceded by a NEWLINE character).

Getlin returns EOF when it encounters an end-of-file, and otherwise returns
the line length (excluding the EOS).

Interspersed calls to getlin and getch are allowed and work properly.

If the file contains characters in a representation other than ascii, the char­
ters are mapped (via inmap) into their internal ascii representation.

Getlin generally assumes a maximum size of the array line passed to it (MAX­
LINE). If the input line exceeds the limit, no NEWLINE character is returned
and the remainder of the line is returned on subsequent calls to getlin.

Getlin and getch are compatible; that is, interspersed calls to getlin and
getch are allowed and work properly. A common implementation is to have
getlin call getch until a NEWLINE character is found (or MAXLINE is reached).

Getlin is able to recognize end-of-file marks from both terminals and files.

SEE ALSO
getch(lib), putch(lib), putlin(lib)

DIAGNOSTICS
None
NAME
getnow - determine current date and time

SYNOPSIS
call 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 the seven element integer
array:

now(1) - year (e.g. 1984)
now(2) - month (e.g. 9)
now(3) - day (e.g. 25)
now(4) - hour (e.g. 13)
now(5) - minute (e.g. 39)
now(6) - second (e.g. 14)
now(7) - millisecond (e.g. 397)

The information returned by getnow() can be used directly or with fmtdat() or
wkday(). 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.

BUGS/DEFICIENCIES
Some systems cannot obtain all the time information described. Array ele­
ments that cannot be filled default to zero.

SEE ALSO
date(tool), fmtdat(lib), wkday(lib)

AUTHOR
STUG basic tape
NAME
getpat - prepare regular expression for pattern matching

SYNOPSIS
len/ERR = getpat( exp, pat )

integer getpat, pat(MAXPAT), len character exp()

DESCRIPTION
Getpat is used to translate a regular expression into a format convenient for
subsequent pattern matching via 'match' or 'amatch'.

A typical scenario for pattern-matching might be:

if ( getpat( regular_expression, pattern ) != ERR )
    stat = match( input_line, pattern )

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 function returns the number of integers in
"pat"; otherwise it returns ERR.

Getpat is essentially a call to makpat with the following parameters:

getpat = makpat( exp, 1, EOS, pat )

SEE ALSO
makpat(lib), match(lib), amatch(lib)
NAME
getsub - generate substitution pattern

SYNOPSIS
indx/ERR = getsub( arg, sub )

integer indx, getsub character arg(), 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(lib)

DIAGNOSTICS
If an error occurs in the encoding, a value of ERR is returned.
NAME

gettyp - get type of file

SYNOPSIS

ASCII/LOCAL/BINARY = gettyp( name )

integer gettyp, type
character name()

DESCRIPTION

'Gettyp' determines whether the file given by 'name' contains ascii characters, local characters (if different from ascii), or binary. The type is returned as ASCII, LOCAL, or BINARY in the functional call. If the file is empty or new, the default file type (generally ASCII) is returned.

The string 'name' contains the name of the file to be checked for type.

'Gettyp' is called by the archiver to store a file's type in the archive header (for informational purposes only). If it is impossible to implement 'gettyp' on a particular system the call to it in the archiver may simply be left out.

The shell also uses 'gettyp' to determine whether a command verb given by the user represents a script file or an executable tool. If the file turns out to be a character (i.e. script) file, the shell then spawns itself with the file as input. Thus, again, if 'gettyp' could not be reliably implemented on a particular system, the user would have to specifically execute his/her script files by:

% sh script ...

SEE ALSO

open(lib), ar(tool)

DIAGNOSTICS

'Gettyp' calls 'open' to obtain the file descriptor. ERR is returned if the file descriptor is incorrect.
NAME
  getwrd - get non-blank word from character array

SYNOPSIS
  size = getwrd( in, i, out )

  integer getwrd, size, i character in(), out()

DESCRIPTION
  Starting at position 'i' in array 'in', 'getwrd' skips any leading blanks and
  tabs and returns the next word and its length. A word is any series of charac­
  ters 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(lib)

DIAGNOSTICS
  None
NAME

gitoc - convert integer to any radix string

SYNOPSIS

number = gitoc( int, str, size, base )

integer gitoc, number, int, size, base character str(size)

DESCRIPTION

'Gitoc' will convert an integer to character string representation in any radix from 2 to 16 (inclusive). The integer to be converted may be considered as either signed or unsigned.

'Int' is the integer to be converted; 'str' is a character array into which the string representation will be stored; 'size' is the size of 'str'. The absolute value of 'base' is the conversion radix. If 'base' is negative, then 'int' is treated as an unsigned number; otherwise, 'int' is considered to be signed. If the specified radix is not in the range 2:16, then a decimal conversion is performed.

For a signed conversion, if the integer is less than zero, its absolute value is preceded by a minus sign in the converted string; a positive number is never preceded by a sign.

The function return is the number of characters required to represent the integer. 'Gitoc' uses a typical divide-and-remainder algorithm to perform the conversion; that is, a digit is generated by taking the remainder when the integer is divided by the radix. For signed conversions, the absolute value of the number is first taken, the digits generated, and the minus sign inserted if needed. For unsigned conversions, the least significant bit of the number is saved, and then the number is shifted right one bit position to put it into the precision range of a standard integer (and effectively dividing the unsigned number by 2). Then, as each digit value is generated, it is doubled and added to the carry from the previous digit position (with the initial carry being the saved least significant digit) and a new carry value is generated.

ARGUMENTS MODIFIED

str

BUGS/DEFICIENCIES

It is suspected that this routine will not work properly on one's-complement machines. Also note that it depends on the MAX_INT definition in the Software Tools library to mask off the sign bit of a word.
SEE ALSO
itoc (lib), other conversion routines ('cto?*' and '?*toc') (lib)

AUTHOR
STUG basic tape
NAME
   glname - get user's login name

SYNOPSIS
   call glname( buf )
   character buf()

DESCRIPTION
   Glname returns the current user's login name. The name is returned in 'buf',
   a character string.

SEE ALSO

DIAGNOSTICS

BUGS/DEFICIENCIES
NAME
gowner - get users owner name

SYNOPSIS
call gowner( buf )

character buf()

DESCRIPTION
Gowner returns the owner name of the current user in 'buf', a character array. The owner name is machine dependent.

The owner name in gowner may later be used to restore the owner name with a call to siowner.

SEE ALSO
siowner(lib)
NAME
  gtime - get current time

SYNOPSIS
  call gtime(time)

    integer time

DESCRIPTION
  'Gtime' returns the current date into 'time', a character string, in the for­
  mat:

    hh:mm:ss.hh

  On some machines, the hundredths of a second are omitted.

  The hour is given as in a 24 hour clock.

SEE ALSO
  date(tool), gdate(lib), ar(tool)
NAME

guname - get users real name

SYNOPSIS

call guname( buf )

character buf()

DESCRIPTION

Guname returns the full name of the current user. The name is returned in the character array 'buf', terminated with the EOS character.

BUGS/DEFICIENCIES

On the Vax, the user's real name is currently returned in lower case.

SEE ALSO

gowner(lib)
NAME
hislb - history prompting library

SYNOPSIS
len/EOF= hprompt( prompt_string, response_buf, fd )

integer hprompt, fd character prompt_string(), response_buf

DESCRIPTION
Hislb is a history prompting library. It consists of a single function, to be
used as a prompt routine. The string given in prompt_string is printed to the
stream given, and a line is read from that stream and placed in
response_buf. The function returns the line length of response_buf or else
returns EOF. However, certain lines of input are intercepted by hprompt,
and are used to manipulate a list, or history, of the commands typed so far.
There are two types of special input: history commands and history substitu­
tions. History commands are used for manipulating the history list and do
not return to the calling program. History substitutions are replaced by the
appropriate previous line from the fd, and are then returned just as if they
had been typed by the user. The history keeps the previous 50 commands.
Any beyond that are lost.

The commands are:

1. History commands

For a history command to be recognized, it must be the only thing on a line.
Whitespace is allowed after commands, as well as between any arguments.

A. History - "!h [n]"

This command prints out the history list. If no argument is given, the entire
history is printed, otherwise, just the last [n] commands are listed.

B. History - "!i"

This command does the same thing as "!h20"

2. History substitutions

History substitutions occur only at the beginning of a line. If there is an aux­
illary command, it must occur immediately after the history substitution
and be the last thing on the line.

A. Repeat last line - "!!"

The sequence "!!" is replaced with the text of the last line typed.
B. Absolute or relative line - "!-[0-9]+", "![0-9]+"

This is replaced with the text of the specified line. For example, "!-4" is replaced with the text of the fourth to last line you typed, and "!4" is replaced with the fourth line typed since starting the calling program.

C. Search for beginning of line pattern - "!<pat>"

This sequence is replaced with the text of the last line that matches the given pattern at the beginning of a line. <pat> is a standard software tools regular expression.

3. Auxiliary commands.

Auxiliary commands are specified by following any of the history substitutions given in (2) above with a ':' and the auxiliary commands. These commands may be in any order, for example, !!:s/a/b/gp !!:gs/a/b/p and !!:pgs/a/b all do the same thing. Auxiliary commands must be the last things in a line, and they are not case sensitive (except for the patterns used in the substitute command.)

A. Print - ":p"

The presence of the 'p' aux commands causes the line typed in to be printed and logged into the history, but not returned to the calling program. This is useful if you desire to perform more than one substitution on a line, or just want to make sure you have the right line before you execute it.

B. Substitute - "':s/<pat>/[<text>[/]]"

The same syntax and effect as for ed(tool).

C. Global - "':g"

Specifying the global auxiliary makes the substitute command ("B" above) be done globally. It is not an error to specify global without a substitute command, or to specify it more than once.

SEE ALSO

sh(tool) - currently uses hislb.
ed(tool) - model for substitute command syntax.

AUTHOR

Marshall Spight
NAME
  index - find position of character in string

SYNOPSIS
  loc = index( str, c )

    integer index, loc character str(), c

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

DIAGNOSTICS
  None
NAME
initst - initialize all standard files and common variables

SYNOPSIS
call initst

DESCRIPTION
This routine is the first routine called by any program desiring to use the ratfor primitives. It is automatically included in any program using the DRIVER macro. It opens STDIN, STDOUT, and ERROUT files, performing any file substitutions necessary. It also prepares the list of arguments needed by getarg and sets up any buffers, variables, etc. needed by the ratfor primitives.

'Initst' initializes any common blocks, variables, buffers, arrays, or whatever is necessary to allow the other ratfor primitives to operate. It may also have to retrieve (via 'makarg') the list of command arguments passed to the program, if this is not automatically available from the operating system.

'Initst' is also responsible for parsing the command line to determine if there have been any file substitutions for STDIN, STDOUT, or ERROUT. The appropriate files (either the user's terminal or the substitutions) are then opened and properly positioned. Arrangements are made so that 'getarg' won't pick up standard file substitution flags on subsequent calls (probably by a call to 'delarg').

SEE ALSO
endst(lib), getarg(lib), delarg(lib), termin(lib), trmout(lib), rat4(tool)

DIAGNOSTICS
If initst cannot function for some reason, the program generally aborts (possibly without an error message since standard output files may not have been opened).
NAME
isatty - determine if file is a teletype/CRT device

SYNOPSIS
YES/NO = isatty( int )

integer isatty, int

DESCRIPTION
This function returns YES if the file specified by 'int' is a teletype-like device, otherwise it returns NO. 'Int' is a file identifier returned by a call to open or create.

When a file is opened, a flag is usually set indicating what device the file is associated with. This function generally reads that flag. Other implementations are possible, depending upon the operating system involved.

'Isatty' is generally used by the tools to determine whether they should issue prompts or not.

SEE ALSO
open(lib), create(lib)

DIAGNOSTICS
NO is returned if 'int' is in error.
NAME
  itob - convert integer to string of base b

SYNOPSIS
  call itob( n, string, size, b )

                integer n, b character string(size)

DESCRIPTION
  Converts the number 'n' to base 'b', and puts the result in 'string', a string of
maximum size 'size'. The base may be anywhere from base two to base thirty-six. If the base 'b' is not valid, the result is given in base ten. If the number is too large to fit in the string, the string is truncated on the left.

SEE ALSO
  putb(lib)
NAME

itoc - convert integer to character string

SYNOPSIS

length = itoc( int, str, size )

integer itoc, length, int, size character str()

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 of the resulting character string, not including the EOS marker. Characters are stored in ascii character arrays terminated with an EOS marker.

SEE ALSO

ctoi(lib), putdec(lib)

DIAGNOSTICS

None
NAME
itoczf - converts an integer to a character string (zero filled)

SYNOPSIS
call itoczf( n, buf, width )

integer n, width character buf()

DESCRIPTION
Itoczf takes the integer n and puts it in base 10 representation in the character array buf padded with leading zeros. The total width of the resulting character string is given by the integer width.

SEE ALSO
itoc(lib)
NAME
length - compute length of string

SYNOPSIS
n = length(str)

integer length, n character str()

DESCRIPTION
Computes the length of a character string, excluding the EOS. The string, 'str', is an ascii character array terminated with an EOS marker. The length is returned by the function.

FILES
None

SEE ALSO

AUTHOR

DIAGNOSTICS
None

BUGS/DEFICIENCIES
NAME
lower - convert string to lower case

SYNOPSIS
call lower(str)

character str()

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.

(This routine is exactly the same as 'fold'.)

SEE ALSO
clower, cupper, upper, fold

DIAGNOSTICS
None

AUTHOR
STUG basic tape
NAME
makpat - prepare expression for pattern matching

SYNOPSIS
len/ERR = makpat( arg, from, delim, pat )

integer makpat, from, pat(MAXPAT) character arg(ARBIT), delim

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 tutorial 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
ggetpat(lib), match(lib), amatch(lib)

DIAGNOSTICS
A value of ERR is returned if a failure occurs in the encoding.

AUTHOR
Whei-Ling Chang

BUGS/DEFICIENCIES
NAME
maksub - make substitution string

SYNOPSIS
indx = maksub( arg, from, delim, sub )

integer indx, maksub, from character arg(), sub(MAXPAT), delim

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 substring (those of the form @1 ...
@9). It also handles escaped characters (@c).

SEE ALSO
getsub(lib), ed(tool)

DIAGNOSTICS
A value of ERR is returned if the encoding fails for any reason.
NAME
  manmac - macros for programmer's manual entries

SYNOPSIS
  (see Request Summary)

DESCRIPTION
  Manmac is a set of roff macros used to simplify the production of Program-
  mer Reference Manual entries. It is generally used by the script m (format
  manual entry). It contains one macro for each of the normal pieces of a
  Manual entry together with the paragraph and display macros from rtsgmac
  (see the attached request summary).

FILES
  /usr/lib/manproto (a prototype manual entry)

SEE ALSO
  manual(info), m(tool), roff(tool), rtsgmac(lib)
NAME
match - match pattern anywhere on a line

SYNOPSIS
YES/NO = match( lin, pat ) character lin(ARB) integer match, 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 pat-
ttern was found anywhere in the line, NO otherwise.

The pattern in 'pat' is a standard Software Tools encoded regular expres-
sion. 'Pat' can be generated most conveniently by a call to the routine
'makpat'.

'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.

SEE ALSO
amatch(lib), makpat(lib), maksub(lib), onsub(lib), find(tool), ch(tool),
ed(tool)
NAME
  mkuniq - get scratch file name based on 'seed'

SYNOPSIS
  call mkuniq( seed, name )

  character seed(), name()

DESCRIPTION
  'Mkuniq' is used to generate scratch file names. 'Seed' is passed as an array
  generally containing one to three characters to be used in generating the
  scratch file name. 'Name' is returned as a scratch file name to pass to an
  'open' or 'create' call.

  'Mkuniq' is used to avoid conflicts which occur when more than one user is
  logged in under a single user or directory name.

SEE ALSO
  open(lib), create(lib)

BUGS/DEFICIENCIES
  The optimum implementation would be to return an absolutely unique file
  name based on 'seed'. However, on most systems this is impossible.
NAME

mtlb - mag-tape library

SYNOPSIS

OK/ERR = mtend( vmssts, files )
OK/ERR = mtrwnd( vmssts )
OK/ERR = mttype( vmssts, type, wsts )

integer mtend, mtrwnd, mttype, type integer files, vmssts, type, wsts

DESCRIPTION

Mtend - Position the tape on drive 'MT' to the end of information on the tape. The end of information is defined to be two end of files in a row. The tape is positioned immediately before the second end of file.

The number of files seen on the tape is returned in 'files'.

If mtend was unable to skip to the end of information, it returns ERR, and vmssts will contain a descriptive error code. Otherwise, mtend returns OK, and vmssts isn't defined.

Mtrwnd - Rewind the tape on drive 'MT'. If mtrwnd was unable to rewind the tape, it returns ERR, and vmssts will contain a descriptive error code. Otherwise, it returns OK, and vmssts isn't defined.

Mttype - Find out what type of tape is on drive 'MT', and whether or not the user can write on it.

The type of tape 'type' may be UNUSED_TAPE if the tape has never been used before, INITIALIZED_TAPE if the tape has just been initialized, BACKUP_TAPE if the tape is a backup tape, QUESTIONABLE_BACKUP_TAPE if the tape might be a backup tape, ANSI_LABELED_TAPE if the tape is organized with volume headers and file headers and trailers in ansi-standard form, and OTHER_TAPE if mttype doesn't have any idea what's on the tape. The best mttype can do really is make an educated guess, but the guess is generally fairly reliable.

'Wsts' will be YES if the user can write on the tape, and NO if the tape is write locked.

If mttype is unable to read the tape for some reason, if returns ERR, and vmssts will contain a descriptive error code. Otherwise, mttype returns OK and vmssts isn't defined.
FILES
 incl/mlib - the include file you must include to use this library

SEE ALSO
 mtr(tool), mtw(tool), back(tool)

DIAGNOSTICS

AUTHOR
 Todd Hammond

BUGS/DEFICIENCIES
NAME
nmfile - get name of file associated with file descriptor

SYNOPSIS
OK/ERR = nmfile( fd, file )

character file() integer fd, nmfile

DESCRIPTION
Nmfile returns the name of the file associated with the file descriptor 'fd' into 'file', a character string which should be of size at least FILENAMESIZE.

If fd is not open, if the name of the file associated with fd is not known, or if fd is not associated with a file (i.e. if fd is associated with a shared memory region), nmfile returns ERR; otherwise it returns OK.

Note again that nmfile is not guaranteed to work for all file descriptors. (In particular it is particularly likely not to work with the fd 'STDIN'.) Use at your own risk.

SEE ALSO
files(info), open(lib), close(lib)
NAME
note - determine file position of next record

SYNOPSIS
OK/ERR = note( offset, fd )

integer note, offset(2), fd

DESCRIPTION
Note() is used in conjunction with seek(). The routine is used to pick up the file position of the next record to be read/written on the file. It 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. 'Offset' is a two-element integer array in which is stored a character count, word address, block and record address, or whatever is appropriate for the operating system in use. Offset should be retained (untouched) by the user and passed to seek() when access to the record is desired. Note() is compatible with whatever implementation that is chosen for seek and the opening of files at READWRITE access. In the editor, note is called to locate the end of file for subsequent writes. Note() returns OK or ERR.

FILES
None

SEE ALSO
seek(lib), ed(tool)

AUTHOR

DIAGNOSTICS
None

BUGS/DEFICIENCIES
NAME
open - open an existing file for reading/writing

SYNOPSIS
int/ERR = open( name, access )

integer int, open, access character name()

DESCRIPTION
Open attaches an existing file to a running program and associates the external
file name with an internal identifier which is then usable by the program.
The file is positioned at the beginning. Opening a fresh instance of an already
open file is permissible.

"name" is a character string representing a pathname or filename in whatever format is used by the local operating system. Name is passed as an
array of ascii characters terminated with an EOS marker.

"access" is a descriptor for the type of access desired - READ, WRITE, READWRITE, or APPEND.

"int" is returned as an integer descriptor to the file.

The file is positioned at the beginning unless APPEND access is requested, in
which case the file is prepared for extension.

Open connects the file to the running program and does what manipulations
are necessary to allow reading and/or writing to the file. An internal descriptor (usually an integer) 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 io buffer for the file. It may also have to
do an initial read to determine the file type (character 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
isatty and gettyp functions.

Open is generally taught to read characters of ascii type as well as local
character type (if not ascii). Translation of characters from local to ascii is
done when the characters are passed to getch and getlin.

Opening a fresh instance of an already opened file is permissible and does
not affect the position of the file as accessed by subsequent or previous calls.

There is generally a limit to the maximum number of files open at any one
time. None of the tools require more than 6.

READWRITE access may cause problems. The only tool which needs this
access is the editor. If necessary you may have to implement it by opening
the file twice--once at read and once at write access.

SEE ALSO
create(lib), close(lib), remove(lib), isatty(lib), gettyp(lib), putch(lib),
getlin(lib), putlin(lib)

DIAGNOSTICS
Open returns ERR if the file does not exist, if one of the necessary directories
(if any) does not exist or is unreadable, if the file is not readable/writeable,
or if too many files are open.
NAME
panic - output a message and stop

SYNOPSIS
  call panic( msg )

DESCRIPTION
Panic writes out the passed message and halts the program. 'Msg' is a character array, and is put onto the file ERROUT.

Panic should only be used if there is some very serious problem with the internal consistency of the tools libraries so that a call to the routine 'error' might not work.

The exact behavior of 'panic' is system dependent.

SEE ALSO
  error(lib)

BUGS/DEFICIENCIES
  On the Vax, a call to panic may not work if there is a problem in the I/O library.
NAME
printf - formatted output to STDOUT

SYNOPSIS
   call printf( fmt, arg1, arg2, ... )

   character fmt()

DESCRIPTION
Printf() converts, formats, and prints its arguments under control of a for­
mat specification, and writes them to standard output (STDOUT). It allows the
user to specify a format string, "fmt", to control the output, and a number of
items to be output: arg1, arg2, etc.

The format string contains literal characters to be printed, as well as formatting
control structures. Formatting control structures consist of a percent
sign, '%', followed by characters describing the action to be performed on
the next argument in the argument list. Characters in the format string that
are not associated with a format control structure are output to the file
without change.

The format string is described in detail in the manual entry for prints(lib).

EXAMPLES
To print a string, followed by a new line ("@n"):

   call printf( "%s@n", str )

To print two real numbers, the first in a field of five (5) with precision two (2),
and the second in a standard field, along with some text for identification,
followed by a newline:

   call printf( "x = %5.2g, y = %g@n", xcoord, ycoord )

SEE ALSO
fprintf(lib), prints(lib)

AUTHOR
Rick Corona

BUGS/DEFICIENCIES
The format string may be a ratfor string or, a quoted or hollerith string. For
example, use of quoted or hollerith strings will work on a Vax, but not on the
Modcomp. It is recommended that hollerith strings not be used if programs
are to be truly portable.

There is a limited number of arguments that can be formatted, depending on
the machine being used. The current RTSG machines limit is seven (7)
arguments.
NAME
prints - formatted conversion into a character array

SYNOPSIS
call prints( dst, fmt, arg1, arg2, ... )

DESCRIPTION
Prints() converts and formats its arguments under control of a format specification. It allows the user to specify an array on which to write the arguments, a format to control the output, and any number of items to be printed. In prints(), "dst" refers to a character array onto which the items should be appended, "fmt" is a format string (discussed below), and the remaining arguments (zero or more) are items to be output according to format control.

The format string contains literal characters to be printed, as well as formatting control structures. Formatting control structures consist of a percent sign '%', followed by characters describing the action to be performed on the next argument in the argument list. Characters in the format string that are not associated with a format control structure are output to the file without change.

Following the '%' there may be:

- An optional minus sign which signifies left adjustment of the converted argument in its field.

- An optional digit string specifying a minimum field width. The converted number will be printed in a field at least this wide and wider if necessary. If the converted argument has fewer characters than the field width it will be padded on the left (or right if the left adjustment indicator has been given) to make up the field width. The padding character is usually a blank; however if the field width is specified with a leading zero, the padding character will be the digit zero.

- An optional period which separates the field width from the next optional digit string - the precision. If a precision is specified, the period is required. Precision specifies the maximum number of characters to be printed from a string, or the number of digits to be printed to the right of the decimal point.

- An optional digit string specifying the precision. (To be used for real number notation and character strings.) For reals, the precision specifies the maximum number of digits to be output following the decimal point; for character strings, the precision specifies the maximum number of characters to be printed from a given string.
- A character which indicates the type of conversion to be applied.

The conversion characters and their meanings are:

- **d** The argument is taken as an integer and is converted to decimal notation.

- **o** The argument is taken as an integer and is converted to octal notation.

- **x** The argument is taken as an integer and is converted to hexadecimal notation.

- **c** The argument is taken to be a single character.

- **s** The argument is taken to be a character string; characters from the string are printed until an EOS is found or until the number of characters indicated by the precision specification is reached.

- **g** The argument is taken to be a real number. Decimal or scientific notation, depending on the magnitude of the number, will be used; non-significant zeros are not printed. (Refer to manual entry for rtoc(lib).)

If the character after the '%' is not a valid conversion character, that character is simply echoed. This means that "%%" may be output with the sequence "%%".

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width.

**EXAMPLES**

To append a string, "str" followed by a new line ("@n") onto a character array:

```
call prints ( buf, "%s@n", str )
```

To output one real number in a field of five (5) characters with a precision of two (2), and a second real number, along with some text for identification, followed by a newline:

```
call prints ( buf, "x = %5.2g, y = %g@n", xcoord, ycoord )
```

To convert an integer to characters and save it in an array (the array will be terminated by an EOS):

```
call prints ( array, "%d", int )
```
SEE ALSO
printf(lib), fprintf(lib)

AUTHOR
Rick Corona

BUGS/DEFICIENCIES
The formal string may be a ratfor string or, on some systems, may also be a quoted or hollerith string. For example, use of hollerith strings will work on a Vax, but not on the Modcomp. It is recommended that hollerith strings not be used if programs are to be truly portable.

There is a limit to the number of arguments to be formatted, depending on the machine being used. The current RTSG machines limit is seven (7) arguments.

The Unix printf() "%f" spec for floating point numbers is currently unimplemented.

Precision is ignored for the "%g" format.

This package cannot handle double precision arguments.
NAME
prompt - prompt user for input

SYNOPSIS
len/EOF = prompt( str, buf, fd )

integer len, prompt character str(ARB), buf(MAXLINE) filedes fd

DESCRIPTION
Prompt determines if "fd" refers to a teletype-like device and, if so, writes
the prompt string "str" to the TTY, and flushes its output buffer to insure the
prompt is printed. A line of input is then read from fd by "getlin". The func­
tion value returned by getlin is also returned by prompt.

No carriage return/line feed sequence is done unless specified by a NEWLINE
in the prompt string. The version of 'prompt' on the tape is essentially:

if ( isatty( fd ) == YES )
{
    call putlin( str, fd )
    call flush( fd )
}
stat = getlin( buf, fd )
return( stat )

Note that prompt expects to be able to read from and write to 'fd'. If this is
not possible on your system, modify prompt to open a separate channel to
the teletype for the write.

SEE ALSO
putlin(lib), remark(lib), flush(lib), isatty(lib)

DIAGNOSTICS
None
NAME
prtnl - print lines to STDOUT with pauses

SYNOPSIS
'Q'/'@N'/EOF = prtnl( rfi, nl, tt )

character prtnl integer rfi, nl, tt

DESCRIPTION
Prtnl will print all lines from the file given by 'rfi' to STDOUT, pausing after every 'nl' lines and prior to printing lines beginning with form feeds. During the pause, the user will be prompted for instructions from the device given by 'tt'.

If STDOUT is not a terminal device, no pausing will occur.

The status returned by prtnl will be EOF if the response to the last prompt was an end-of-file, the letter 'Q' if the response to the last prompt was a 'q' or 'Q', or a newline if the file was printed completely.

SEE ALSO
fcopy(lib)
NAME
putb - write out number in base b

SYNOPSIS
  call putb( n, width, b, stream )

  integer n, width, b, stream

DESCRIPTION
Writes the number n in base b on the stream 'stream'. 'Width' is the
minimum width of the number written out. If 'width' is not large enough to
contain the number, more space is used.

SEE ALSO
  itob(lib)
NAME
putc - put character onto STDOUT

SYNOPSIS
call putc( char )

character char

DESCRIPTION
Putc writes the passed character on STDOUT. 'Char' is of type character.

SEE ALSO
putc(lib)

NAME
putch - write character to file

SYNOPSIS
   call putch( c, int )

   character c integer int

DESCRIPTION
   Putch writes the character c onto the file specified by int. 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; however, if the
   output file is not of ascii type the necessary conversion is done.

   Interspersed calls to putch and putlin work properly. One implementation is
to have putlin perform repeated calls to putch.

SEE ALSO
   putlin(lib), getch(lib), getlin(lib), putc(lib)

DIAGNOSTICS
   None
NAME
putdec - write integer in specified field width

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(lib)

DIAGNOSTICS
None
NAME
putint - output integer in specified field

SYNOPSIS
call putint(n, wid, fd)

integer n, wid, fd

DESCRIPTION
Putint converts the integer n to a character string, and then writes the
string on file 'fd', using the integer 'wid' as the width. The string is padded
with leading blanks, if necessary.

SEE ALSO
putb(lib), putdec(lib)
NAME
   putlin - output a line onto a given file

SYNOPSIS
   call putlin( line, int )

   character line() integer int

DESCRIPTION
   Outputs the character array line onto the file specified by int.

   The message array is generally a Ratfor double quoted string. It may also be
   an ascii character array terminated with an EOS marker.

   Putlin and putch are compatible; that is, interspersed calls to putlin and
   putch are allowed and work properly. A common implementation is to have
   putlin call putch until an EOS marker is found.

SEE ALSO
   putch(lib), remark(lib), getch(lib), getlin(lib)

DIAGNOSTICS
   None
NAME
putspaces - write a specified number of blanks to STDOUT

SYNOPSIS
    call putspaces( n )

    integer n

DESCRIPTION
    Putspaces outputs n BLANK characters to STDOUT. If n is less than or equal
to zero, no BLANK characters are output.
NAME
putstr - output character string in specified field

SYNOPSIS
call putstr( str, wid, fd )

character str() integer wid, fd

DESCRIPTION
Putstr writes the character string str onto the file 'fd' using the integer width 'wid'. If 'wid' is positive and the str is less than 'wid', then 'wid' BLANK characters are output before the string. If 'wid' is negative, then '-wid' BLANK characters are output after the string.

SEE ALSO
putlin(lib), remark(lib)
NAME
queld - integer queue and stack library

SYNOPSIS

call queinit(qp)
call queclear(qp)
call quedestroy(qp)
isempty = queempty(qp)
call quefinsert(qp, int)
call quebinsert(qp, int)
int = quefremove(qp)
int = quebremove(qp)
int = quefront(qp)
int = queback(qp)
qlp = queloopinit(qp)
another = quenext(qlp, int)

call stkinit(sp)
call stkclear(sp)
call stkdestroy(sp)
isempty = stkempty(sp)
call stkpush(sp, int)
int = stkpop(sp)
int = stktop(sp)

integer qp, int, qlp, sp
logical isempty, queempty, another, quenext, stkempty
integer quefremove, quebremove, quefront, queback
integer stkpop, stktop, queloopinit

DESCRIPTION

These routines implement a simple double-ended queue package. A double-ended queue is a sequence of items. Only the items on the two ends can be examined or modified.

In this package, the items stored are integers. If you want to store other types of items in a queue, you can store pointers to them, since pointers are integers.

queinit - initialize a queue. This routine creates a new, empty queue, and gives you a pointer to it. The pointer is used by all of the other queue routines.

queclear - clear a queue. This routine removes all of the integers from a queue and throws them away. The queue still exists, but it is empty.
quedelete - deallocate the storage for a queue. This routine not only removes all of the integers from a queue, it also destroys the queue itself and sets the pointer you passed it to NIL.

queempty - tell whether a queue is empty or not. If the queue has any integers in it, queempty returns .false.; if there are no integers, it returns .true..

quefinsert - insert an integer onto the front of a queue. You pass this routine a queue pointer and an integer, and it places the integer on the front end of the queue.

quebinsert - insert an integer onto the back of a queue. You pass this routine a queue pointer and an integer, and it places the integer on the back end of the queue.

quefremove - remove an integer from the front of a queue. This routine gets the integer on the front end of a queue, removes it, and returns it to the caller.

quebremove - remove an integer from the back of a queue. This routine gets the integer on the back end of a queue, removes it, and returns it to the caller.

quefront - examine the integer on the front of a queue without disturbing it. This routine gets the integer on the front end of a queue and returns it to the caller. The integer is NOT removed.

queback - examine the integer on the back of a queue without disturbing it. This routine gets the integer on the back end of a queue and returns it to the caller. The integer is NOT removed.

queloopinit - prepare for a loop through the elements of a queue. This routine is used to initialize a for-loop through the elements of a queue. quenext is used to do the actual looping.

quenext - generate the next element in a queue loop. This routine is used to get the next element in a loop through the elements of a queue. The loop pointer must have been initialized by queloopinit. If there is another element, quenext gets it and returns .true.; if there are no more elements, quenext returns .false. The intended use of queloopinit and quenext is:

```
for ( qlp = queloopinit( qp ); quenext( qlp, int ); )
{ ... }
```

In addition to the queue routines, there are the following routines to manipulate stacks. A stack is a restricted queue - only the item on the top can be examined or modified.
stkinit - initialize a stack. This routine creates a new, empty stack, and gives you a pointer to it. The pointer is used by all of the other stack routines.

stkclear - clear a stack. This routine removes all of the integers from a stack and throws them away. The stack still exists, but it is empty.

stkdestroy - deallocate the storage for a stack. This routine not only removes all of the integers from a stack, it also destroys the stack itself and sets the pointer you passed it to NIL.

stkempty - tell whether a stack is empty or not. If the stack has any integers in it, stkempty returns .false.; if there are no integers, it returns .true..

stkpush - push an integer onto a stack. You pass this routine a stack pointer and an integer, and it places the integer on the top of the stack.

stkpop - pop an integer off of a stack. This routine gets the integer on the top of a stack, removes it, and returns it to the caller.

stktop - return the value on top of a stack without popping it. This routine gets the integer on the top of a stack and returns it to the caller. The integer is NOT popped.

Quelb also has an optional statistics-gathering feature. It keeps a count of the number of queue headers and nodes allocated and freed. If you suspect your program is forgetting to free some of its queues, and therefore running out of memory, this feature can be very useful.

The statistics-gathering code is conditionally-compiled out of the installed quelb. To use it, copy the source of quelb into your space, change the definition of STATISTICS from NO to YES, and compile it. Then add a call to questats to your program where you want the statistics printed. They will be written to ERROUT.

AUTHOR
Jef Poskanzer

BUGS/DEFICIENCIES
The usage statistics code is slightly non-portable. It assumes a local variable is static.
DIAGNOSTICS

- insufficient memory. The dynamic memory allocation routine (malloc) failed.

- non-existant queue. The queue pointer that you passed to the routines did not in fact point to a queue.

- internal error - inconsistent queue detected. Inform your local "Tools" person.
NAME
query - print command usage information on request

SYNOPSIS
call query( usage )

caller usage()

DESCRIPTION
Many Software Tools 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 user must supply the usage string in 'usage'. For example,

    string usage "gronk [-i] file ...

When called, 'query' checks to see that the program 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 program terminates. If not, 'query' returns quietly.

The implementation calls getarg to determine if the only argument is a '?' and if so, a call to 'error' is made.

SEE ALSO
error(lib)

AUTHOR
STUG basic tape
NAME
quest - write a prompt and accept the user response

SYNOPSIS
len = quest( prompt, pfd, answer, afd )

integer len, quest, pfd, afd character prompt(), answer()

DESCRIPTION
The Software Tools 'prompt' routine will not work on all machines. 'Quest' is a
machine-independent substitute to this old 'prompt' routine.

Quest will write prompt to the file given by the file descriptor pfd and then
read the users answer into the character array answer from the file given by
the file descriptor afd. NEWLINEs are allowed in the prompt string. The
result returned by quest will be EOF if the answer was an end-of-file; other­
wise it will be the length of the answer (counting the NEWLINE but not count­
ing the EOS).

If pfd is not a terminal device, no prompt is performed. However, the read­
ing of the user answer still occurs.

On prompting, the curser will be left at the end of the prompt string unless a
NEWLINE character is included in the string.

SEE ALSO
prompt(lib), putlin(lib), remark(lib)
NAME
readf - read from an opened file

SYNOPSIS

\[ \text{count} = \text{readf}( \text{buf}, n, \text{fd} ) \]

integer count, readf, n, fd

DESCRIPTION

Readf reads "n" bytes (or words) from the file opened on file descriptor "fd" into the array "buf". The bytes (or words) are placed in "buf" one per array element. Readf is the typical way of doing binary reads on files. Whether buf is declared an integer or a character array is dependent upon which is most appropriate for the host operating system.

Readf returns the number of bytes/words 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. 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(lib), getch(lib), putch(lib), getlin(lib), putlin(lib)

DIAGNOSTICS

None
NAME
remark - print single-line message

SYNOPSIS
call remark( messag )

DESCRIPTION
Remark writes a message onto the standard error file ERROUT. A newline is always generated, even though one may not appear in the message.

The message array is generally a Ratfor double quoted string. It may also be an ascii character array terminated with an EOS marker.

Remark is very similar to error except it returns after printing, instead of stopping. It is also similar to putlin except it guarantees that a newline character is generated after the line is printed and it prints the line to ERROUT instead of STDOUT.

SEE ALSO
error(lib), putlin(lib)

DIAGNOSTICS
None
NAME
  remove - remove a file

SYNOPSIS
  OK/ERR = remove(filename)

  integer remove character filename()

DESCRIPTION
  Remove removes a file from the file system. The status returned is either OK
  if the file existed and remove was able to remove it, or ERR.

SEE ALSO
  open(lib), close(lib), create(lib), rm(tool)

DIAGNOSTICS
  "io2lb$remove: Unknown status from t_erase(), n hex."
  A bizarre error occurred. Notify the system manager.
NAME
   rewind - rewinds file

SYNOPSIS
   call rewind( fd )

   integer fd

DESCRIPTION
   Rewind rewinds the stream fd so that the next i/o operations will take place
   at the beginning of the file.

BUGS/DEFICIENCIES
   If fd is a file description of a file which cannot be rewound (e.g. crt or card
   reader), rewind returns with no error. However, machine-dependent error
   actions may occur.
NAME
rtoc - convert a real number to a character string.

SYNOPSIS
length = rtoc( number, str, size )

integer length, rtoc, size real number character str()

DESCRIPTION
Rtoc converts the real number 'number' to a character string 'str', of length 'size'. The length of the result string (excluding the EOS) is returned as the function value.

The conversion is free format and compatible with 'ctor'. The format adheres to the IEEE Human Factors Society recommendation for free format floating point numbers and, in practice, seems to be well liked. However, the rules may seem a trifle complex:

- if the magnitude of the number is < 10^-6 (6 digits) and it has no fractional part, it is converted without a decimal point or exponent. E.g., the result of '1.234*1000.' would be converted to '1234'.

- if the magnitude of the number is < 10^-6 but has a fractional part, it is converted without an exponent. E.g., the result of '1234./100.' would be converted to '12.34'.

- if the magnitude is >= 10^-6 or < 1 the number is printed with an exponent (and a fraction if necessary). The conversion is done to "Engineering Notation". I.e., the exponent is always a power of 3 (milli, micro, kilo, etc.) and there will be 1, 2 or 3 digits to the right of the decimal point. For example,
  1.234e7 converts to 12.34e6
  0.123 converts to 123e-3

SEE ALSO
ctor(lib)

FILES
None
AUTHOR
Van Jacobson

BUGS/DEFICIENCIES
NAME
   scopy - copy (sub)string to another string

SYNOPSIS
   call scopy(from, i, to, j)

character from(), to() integer i, j

DESCRIPTION
   Copies the (sub)string of 'from', starting in location 'i', into array 'to', start­
ing at 'j'.

SEE ALSO
   stcopy(lib), addset(lib)

DIAGNOSTICS
   None
NAME
sdrop - drop characters from a string (APL-style)

SYNOPSIS
call sdrop( from, to, length )

character from(), to() integer sdrop, 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
cloc, length

SEE ALSO
stake (lib), index (lib), substr (lib)

AUTHOR
STUG basic tape
NAME
seek - move read/write pointer on random-access file

SYNOPSIS
OK/ERR = seek( offset, int )
integer offset(2), int

DESCRIPTION
Seek positions the file specified by int for a subsequent read beginning at offset. The offset is a 2-word array containing information in terms of characters, words, records, block numbers, lines, or whatever is appropriate for the operating system. Offset is originally obtained by a call to note. Once the file is positioned by a call to seek, reading can be done using the standard I/O calls (getch, getlin).

Seek can also be used for seeking to the end of a file and performing a write (thus extending the file). Rewriting in place is generally not allowed.

Seek depends heavily upon the peculiarities of the operating system. It can generally be used on files opened at READWRITE access. The offset units are chosen to be whatever is most appropriate for the system involved.

SEE ALSO
note(lib)

DIAGNOSTICS
None
NAME
  setabort - set abort status

SYNOPSIS
  call setabortstatus

DESCRIPTION
  Setabortstatus sets the abort status to a value that notifies the parent proc­ess that the tool aborted when the tool exists.

  This is useful if a tool detects an error and wants to set an error flag to inform the parent of the error without displaying error messages.

  If the process was executed in WAIT mode, CHILD-ABORTED is returned to the parent process.

SEE ALSO
  error(lib) - error will automatically call setabortstatus. spawn(lib) - how to execute a subprocess

BUGS/DEFICIENCIES
  The parent of a process executed in NOWAIT or BACKGR mode can't find out that the child aborted.
NAME
  shell - shell sort an integer array in increasing order

SYNOPSIS
  call shell( ray, n )
  integer ray, n

DESCRIPTION
  Shell sorts the first n elements of integer array 'ray' in increasing order. N
  is an integer. This means that after calling shell, ray(1) will contain the
  smallest value. Ray(2) will contain the next smallest value, and so on.

SEE ALSO
  sort(lib)
NAME
siowner - set internal owner name

SYNOPSIS
call siowner( name )
character name()

DESCRIPTION
Siowner sets the internal owner name to name, a character array.

The internal owner name determines which files this process can access. It
doesn't affect the files which its parent process can access. On a Modcomp,
an owner name is any sequence of characters. On a Vax, it is a UIC.

BUGS/DEFICIENCIES
Privileges may be required to execute this routine.
NAME
skipbl - skip blanks and tabs at specified position in a string.

SYNOPSIS
   call skipbl( str, i )
   character str() integer i

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(lib)

DIAGNOSTICS
   None
NAME
sleep - go to sleep for a specified time period

SYNOPSIS

    call sleep( seconds )

    real seconds

DESCRIPTION
You specify how long to sleep in seconds. The routine waits that long before
returning.

AUTHOR
Jef Poskanzer

BUGS/DEFICIENCIES
On the VAX, sleep is only accurate to within 1/100th of a second. On the
MODCOMP, sleep is only accurate to within 1/100th of a second.
NAME
spawn - spawn a subprocess

SYNOPSIS
OK/ERR/CHILD_ERROR = spawn( image, args, desc, waitmode )

integer spawn, waitmode character image(), args(), desc(PIDSIZE)

DESCRIPTION
Spawn() is the user visible interface which allows the creation of a sub-

process.

Normally, the character string "image" contains the pathname of the image
to execute. "Image" may be a tools pathname of a tool, a VMS file
specification of a tool, or else the string "local". Thus, there are two kinds of
spawns: normal and local.

For a normal spawn, the character string "args" contains the arguments for
the new child. Arguments are blank delimited, that is a blank character fol-

lows each argument. Escapes are allowed in the argument string, as follows:

(1) When an '@' character is seen, it is removed and the character following
it loses special meaning.

(2) When a single (or double) quote is seen, it is removed and all characters
(including '@' and '@ ') lose their special meaning until a matching sin-
gle (or double) quote is found (and removed).

Otherwise, the blank character delimits arguments. For example, the argu-
ment strings:

    string args "-t 'Frontal View' "
    string args "-t Frontal@ View "
    string args "-t Frontal' 'View "
    string args "-t Frontal' View "
    string args "'-t 'Frontal View' "

all specify the two arguments "-t" and "Frontal View".

It is possible to redirect the standard files for a local spawn. For example, if
you wanted to use the crypt tool to encrypt a file, you might spawn to crypt
with an argument string like:

    string args "keyword <infile >outfile"

The redirection of standard files is inherited by the children of children you
spawn.

For a local spawn, the character string "args" contains a DCL command line;
it must conform to the grammar rules for the VAX/VMS command language.
Normally, a DCL command line begins with a dollar sign. Also, command lines may be continued to the next line by ending a line with a minus sign. It is possible to redirect STDIN, STDOUT, and ERROUT, though you may not redirect STDOUT or ERROUT for append and the file names must be VMS file specifications. When scanning an argument string for file redirections, quotes are not handled specially. When an argument that begins with a '<', '>', or '?' character is found at the beginning of an argument, it is taken as a file redirection argument and removed from the string. Arguments that begin "@<", "@>", or "@?" have the '@' removed and lose their special meaning.

After calling spawn(), the array "desc" contains a process identification descriptor, in this implementation a hex string of the new child's pid.

The integer "waitmode" indicates how the parent wants to wait for the new child. A value of WAIT causes spawn() to delay until the child has exited. A value of NOWAIT causes spawn() to return immediately. A value of BACKG causes the child to be created in the background and requires the DETACH privilege for use. It is important to note that under VAX/VMS, NOWAIT isn't very useful because when a process dies, all of its descendents die too.

When spawn() returns control to the program, the integer "status" contains OK if the spawn was successful. If there was some problem creating the child, ERR is returned. If the child aborts and "waitmode" was WAIT, then CHILD_ERROR is returned.

**DIAGNOSTICS**

**AUTHOR**
Craig Leres

**BUGS/DEFICIENCIES**
In this implementation, local spawns must be WAIT mode.

It is not possible to redirect the output of certain "builtin" DCL commands; for example spawning to the local with the argument:

```
string args "$ show daytime >JUNK."
```

doesn't work as expected.

**SEE ALSO**
NAME
splb - search path library

SYNOPSIS
YES/NO = fullpathname( pathname ) fd = lookforpath( pathname, access, filename )

integer fd, lookforpath, fullpathname, access character filename(), pathname()

DESCRIPTION
Splb is a library of routines that deal with filenames. Here is a description:

lookforpath - This routine is similar to open(), i.e. the file specified by "pathname" is opened for access "access"; but additionally, the full path name of the file that was actually opened is returned in "filename". This is particularly useful for discovering exactly which file on a search path was opened. If the search path file cannot be opened, a system default search path file is used.

fullpathname - This routine determines if a filename is a full path name or not. The following are examples of full path names:

```
./
~/
/~
~user/`
./`
../`
```

The routine returns YES if "pathname" is full, otherwise NO.

SEE ALSO
paths(info), files(info)
NAME
stake - take characters from a string (APL-style)

SYNOPSIS
n = stake(from, to, length)

character from(), to() integer stake, length, n

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'.

SEE ALSO
sdrop(lib), index(lib), substr(lib)

AUTHOR
STUG basic tape
NAME
stcopy - string copy, incrementing index

SYNOPSIS
call stcopy( from, i, to, j )

integer i, j character from(), to()

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(lib)

DIAGNOSTICS
None
NAME
  strcmp - compare two strings

SYNOPSIS
  +1/0/-1 = strcmp(s1, s2)

character s1(), s2() integer strcmp

DESCRIPTION
  If s1 is equal to s2, 0 is returned. If a prefix is equal, but s1 is shorter or the
  trailing characters in s1 are less than s2, then -1 is returned; otherwise +1 is
  returned. This is better known as "dictionary" order.

SEE ALSO
  equal(lib)

AUTHOR
  Stug Tape
NAME
  strim - trim trailing blanks and tabs from a string

SYNOPSIS
  length = strim(str)

  character str() integer strim, length

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. 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.

SEE ALSO
  stake(lib), sdrop(lib), substr(lib)

AUTHOR
  STUG basic tape
NAME
tblib - symbol table routines

SYNOPSIS
OK/ERR = delete( table, symbol )
OK/ERR = enter( table, symbol, def, defsize )
OK/ERR = lookup( table, symbol, def, defsize )
OK/ERR = mktabl( table, defsize, deftype )
OK/ERR = rmtabl( table )
EOF/OK/ERR = sctabl( table, symbol, def, defsize, ind )

0/ind = tbfnd( table, symbol )
0/ind = tbadd( table, symbol, def, defsize )
OK/ERR = tbdel( table, ind )
OK/ERR = tbcopy( table, ind, def, defsize )
OK/ERR = tbncop( table, ind, symbol )
OK/ERR = tbsiz( table, ind, defsize )
0/nchar = tbsiz( table, ind )

DESCRIPTION
"TbLib" is a collection of routines used to maintain symbol tables. The routines are used by tools like "macro", "roff" and "rat4". The following conventions apply to all routine descriptions. "Table" is an integer returned by mktabl to identify a symbol table. It has meaning only to the table routines. "Symbol" is a character string specifying the symbol to be looked up or entered. "Def" is an array containing the definition associated with a symbol or index. "Defsize" is the number of elements contained in a definition of type "deftype". A "defsize" of zero in a "mktabl" call indicates variable length definitions (character strings for example). "Deftype" indicates the type of a definition element. "Deftype" should be specified with one of the following: DS_INTEGER (integer definition), DS_CHAR (character), DS_REAL (real number), or DS_LOGICAL (logical definition). For example, to create a table which will always contain definitions consisting of 5 integers, you would specify:

sts = mktabl( tbl, 5, DS_INTEGER )

The pointer "tbl" represents the symbol table and has meaning only to the other table routines.

In general, access to the symbol table is via "mktabl", "enter", "lookup", and "delete". However, the lower level tblib routines (tbfnd, tbadd, tbdel, tbcopy, etc.) can be called if necessary.

"Mktabl" creates a symbol table for manipulation by the symbol table routines. The symbol table is a general means of associating data with a symbol. "Mktabl" returns the symbol table pointer in the "table" argument. "Table" only has meaning to the other symbol table routines. It must be passed to
the other routines to signal which symbol table is being manipulated. "Defsize" is the number of information, or definition, elements to be associated with each symbol. "Deftype" is the type of each of these definition elements. For example, a "defsize" of 3 and a "deftype" of DS_CHAR would specify that the definitions for each symbol table entry will be 3 characters long. A defsize of zero indicates that the definitions are variable length for this table. However, an explicit type must always be specified.

"Enter" adds a symbol and its definition to the symbol table "table". "Defsize" will be ignored for tables with fixed length definitions (specified by a non-zero defsize parameter in the original mktabl call). For tables with variable length definitions (specified by a zero parameter in the original mktabl call), "defsize" elements will be copied from the "Def" array into the symbol table. If both the table definition size and "defsize" are zero, no definition is copied from "array" into the symbol table. This is not an error. If any entries for "symbol" are already in "table", the old "symbol" entries will first be deleted. If the table is too full for more entries, ERR is returned, otherwise the "sts" returned will be OK.

"Lookup" searches the symbol table "table" for "symbol", and if it is found, copies the associated definition to the array "def". The size of the definition is returned in "defsize". If the symbol is found, "lookup" returns OK, otherwise ERR is returned.

"Delete" removes the (most recent) definition of the symbol "symbol". If there is no definition for the symbol "symbol" or the delete failed, delete returns ERR, otherwise it returns OK.

"Rmtabl" is used to remove a symbol table created by "mktabl". The sole argument is the symbol table pointer, "table", originally created by "mktabl". "Rmtabl" frees all dynamic memory associated with the "table". This means that all pointers previously returned by symbol table routines for "table" are no longer valid.

"Sctabl" provides a way to access all symbols in a specified symbol table. Successive calls to "sctabl" return symbols and their associated definitions and sizes. When "sctabl" returns EOF, the entire table has been scanned. The 4th argument, "ind", 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. If an error occurs during the scan, ERR is returned.

"Tbfind" can also be used to find a symbol in a specified symbol table. However, instead of returning the definition as "lookup" does, "tbfind" returns a pointer, "ind", to the definition. "Ind" can then be used by tbcopy, tbdel, tbncop, tbsiz, and tbdsiz. If the symbol isn't in the table, "ind" will be zero.

"Tbadd" can also be used to add a symbol to a specified table. However it is different than "enter" in that "tbadd" will not delete an already existing
symbol entry. However, "tbfind" will always return the most recent definition of a symbol so the old definition will be inaccessible until the new one is deleted (except though use of it's index number.) A pointer to the newly added symbol table entry is returned unless an error occurs, in which case 0 is returned.

"Tbdel" can be used to delete a symbol from a specified table using an "ind" returned by "tbfind". If "ind" does not refer to something in the symbol table, bizarre program behavior is to be expected. "Tbdel" will attempt to return ERR if the entry cannot be deleted, otherwise OK will be returned.

"Tbcopy" can be used to copy a symbol definition into the array "def" given an index, "ind", returned by "tbfind". The size of the definition is returned in "defsiz". "Tbcopy" will attempt to return ERR if the definition cannot be obtained, otherwise OK will be returned.

"Tbcop" can be used to copy the "symbol" text into the character array. If the symbol cannot be copied, ERR is returned, otherwise OK is.

"Tbnsiz" can be used to obtain the size (not including EOS) of the symbol for the symbol entry described by "ind". "Ind" must have been obtained with a call to "tbfind". If the size cannot be determined, "tbnsiz" will return 0, otherwise it will return the length of the symbol name.

"Tbdnsiz" can be used to obtain the size of the definition for the symbol entry described by "ind". "Ind" must have been obtained by a call to "tbfind". The defsize for all fixed length definitions will be the same as that specified in the "mktabl" call. The defsize for variable length definitions will be the same as that specified in the "enter" or "nmadd" call. If the size cannot be determined, ERR will be returned.

SEE ALSO
roff(tool), macro(tool), rat4(tool)

AUTHOR
Theresa Breckon
NAME
termin - pick up identifier of read channel to users terminal

SYNOPSIS
call termin(name)

DESCRIPTION
'Termin' is used to get the name of the user's terminal so that a channel to it
can be opened (via an 'open' call) for reading.

Most systems have a standard file name associated with a user's terminal.
This routine is used by initr4 to open STDIN, as well as by the roff tool, the
form tool, and the label tool to open a line to the user's terminal for reading.

SEE ALSO
trmout(lib), initr4(lib), open(lib), roff(lib), form(lib)

DIAGNOSTICS
None
NAME
trmout - pick up identifier of write channel to users terminal

SYNOPSIS
call trmout( name )

character name()

DESCRIPTION
'Trmout' is used to get the name of the user's terminal so that a channel to it
can be opened (via an 'open' call) for writing.

Most systems have a standard file name associated with a user's terminal.
This routine is used by initr4 to open STDOUT and ERROUT, and by the shell,
the form tool, and several other tools for prompting the user.

SEE ALSO
termin(lib), initr4(lib), open(lib), sh(tool), form(tool)

DIAGNOSTICS
None
NAME
  trmsiz - get terminal size in lines

SYNOPSIS
  call trmsiz( n )

  integer n

DESCRIPTION
  Trmsiz sets the passed parameter n to the number of lines for the current terminal.

BUGS/DEFICIENCIES
  On the Modcomp, it always assumes the terminal screen has 40 lines. N is returned as ERR if the terminal size cannot be computed.
NAME
   type - determine type of character

SYNOPSIS
   t = type( c )

   character t, c, type

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
   alldig(lib)

DIAGNOSTICS
   None
NAME
typlb - determine the basic type of a file

SYNOPSIS
type = typfil( file )

integer type, typfil character file()

DESCRIPTION
Determines the basic file type of the passed filename. File types are:

SOURCE
BIN
USL
LIB
DIR
NOPEN

FILES
/usr/incl/typlb - definitions of types

SEE ALSO
typeof(tool), gettyp(lib)

AUTHOR
Van Jacobson, Vina Ward, Jef Poskanzer

BUGS/DEFICIENCIES
On the VAX, a library file is classified as binary. This may change in the future.
NAME
upper - convert string to upper case

SYNOPSIS
call upper( str )

character str()

DESCRIPTION
Converts the array 'str' to upper case, if not already upper case. 'Str' is an
ascii character array terminated with an EOS marker.

SEE ALSO
cupper(lib), fold(lib), clower(lib)

DIAGNOSTICS
None
NAME
wclb - routines to match wildcards in filenames

SYNOPSIS
num = expand( name, lptr ) YES/NO = wcmatch( name, pat ) OK/ERR =  
wclstmk( lptr ) OK/ERR = wclstadd( lptr, name ) OK/ERR = wclstrem( lptr, name ) OK/ERR = wclstfree( lptr )

integer num, expand, wcmatch, lptr integer wclstmk, wclstadd, wclstrem, wclstree character
nameO, patO

DESCRIPTION
A filename including any of the characters '*' , '?' , or '[' is treated as a pattern to be matched and replaced by the filenames that match it. The wildcard characters are interpreted as:

* matches any string, including the null string;

? matches any single character;

[...]] matches any of the characters enclosed. A pair of characters separated by '-' matches any character lexically between the pair.

A '.' at the beginning of a simple filename, and '/\', must be matched explicitly.

The routines are as follows:

wcexpand - takes a pathname possibly containing wildcard characters and a pointer to a linked list (created by wclstmk) and returns a list of the file names which match the pattern, in the order that they are found in the directory or directories. The list is presumed to be empty, but if it contains any nodes they will follow the added nodes when the list is returned. Wcexpand also returns the number of names which matched.

wcmatch - takes a simple name and a wildcard pattern for a simple name and returns YES if they match, NO if they do not.

The list handling routines manage a singly linked list of pointers to names.

wclstmk - creates the head of a singly linked list and returns a pointer to it. Status is OK if it was able to allocate the memory, ERR otherwise.

wclstadd - creates a new linked list node, containing the passed name, following the node specified by the passed pointer. Status is OK if it was able to allocate memory for the node and the name, ERR otherwise.

wclstrem - removes the linked list node following the one specified by the pointer and returns the name. Status is OK if a node existed, ERR if the list was
empty.

wclstfree - removes any nodes remaining on the linked list and destroys the list head.

DIAGNOSTICS
Wcexpand will produce an error message and abort if wclstmk or wclstadd is unable to allocate memory.

AUTHOR
Jane Colman

BUGS/DEFICIENCIES
NAME
weofbc - write EOF before closing READWRITE file

SYNOPSIS
call weofbc( rfi )

integer rfi

DESCRIPTION
Weofbc write an end of file on the stream given by rfi, assuming that the next
thing done will be a close.

The implementation of this routines is machine-dependent. On some
machines, the EOF of READWRITE files can be lost and therefore the user has
to insure the EOF placement by using this routine. On other machines this is
not necessary, and this routine will be a no-op. Portable code necessitates
calling this routine before closing any READWRITE files.
NAME
wkday - calculate day of the week

SYNOPSIS
\[ \text{dow} = \text{wkday}(\text{month}, \text{day}, \text{year}) \]

integer month, day, year, dow, wkday

DESCRIPTION
This routine calculates the day of the week. The three arguments completely specify the date:

- month 1-12
- day 1-28, 29, 30, or 31
- year e.g. 1980

The returned value is the day of the week, where 1 is Sunday (and 7 is Saturday).

The current day of the week might be found using:

\[ \text{integer dow, wkday, now(7)} \]
\[ \text{call getnow (now)} \]
\[ \text{dow} = \text{wkday(now(2), now(3), now(1))} \]

Implementated using Zeller's Congruence.

SEE ALSO
getnow(lib)

AUTHOR
STUG basic tape
NAME
writef - write to an opened file

SYNOPSIS
count/ERR = writef( buf, n, fd )

integer count, writef, n, fd

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. How buf is declared is dependent upon what is most appropriate for the host operating system.

Writef returns the number of bytes/words actually written. In most cases, this is equal to "n". If, however, a write error occurs, writef returns ERR. 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(lib), putch(lib), putlin(lib)

DIAGNOSTICS
None

AUTHOR
STUG basic tape
NAME

yyplb - yacc parser library

SYNOPSIS

OK/ERR = yyparse( value ) -1/rule = yyfdrd( state, token ) -1/state = yyfdtr( state, token ) call yydored( rule ) call yydotr( state ) OK/ERR = yyperr( msg ) call yyerror( msg )

integer rule, state, token, value integer yyparse, yyfdrd, yyfdtr, yyperr character msg()

DESCRIPTION

This library contains the parse driver routine for yacc. It is compiled with the parse code generated by yacc from the grammar input file. All of the routines and the global variables in yyplb begin with the letters 'yy'. This is done to avoid conflicts with the code provided by the user in the yacc grammar file.

yyparse - The main program, usually supplied by the user in the grammar file, must call the parse driver routine, yyparse. If the parse was successful, yyparse returns OK, otherwise ERR. The value of the last reduction is returned in 'value'.

yyfdrd - This routine determines if a reduction should be done considering the current 'state' and the lookahead 'token'. If a reduction is possible, the number of the grammar rule to be reduced is returned, otherwise -1 is returned.

yydored - This routine does the reduction for 'rule'. It pops N symbols off the parse stacks, where N is the number of LHS symbols in the grammar rule, 'rule'. Yydored then pushes the RHS symbol of 'rule' onto the token stack, and finds and pushes the goto state onto the state stack.

yyfdtr - This routine determines if a transition, or shift, is possible considering the current 'state' and the lookahead 'token'. If a shift is possible, the state to shift to is returned, otherwise -1 is returned.

yydotr - This routine does a shift, or transition, to 'state'. It pushes 'state' and the lookahead token on to the parse stacks.

yyperr - This is the error-recovery routine. It pops the parse stacks, until a state is found from which it is legal to shift the error token, error. If no such state is found, yyperr returns ERR, otherwise the state is left on top of the state stack and the error token is made the lookahead token, so that the next parse action will be to shift the error token and continue parsing. The parser is put into error-recovery mode until three tokens have been successfully read and shifted. When in error-recovery mode, no parse errors are reported. The error-reporting routine yyerror is
then called, passing 'msg' as the argument.

**yyerror** - this routine simply calls remark to output the passed error message to ERROUT. However, the user may want to provide her own version of yyerror which does a better job of reporting the error.

The declarations for the parse stacks and current parse values are kept in common blocks in the include file `incl/yyypcom`. The declarations for the parse tables generated by yacc are kept in common blocks in the include file `+INCL/yytblc`.

SEE ALSO
yacctut(tutorial), yacc(tool)

FILES
`+INCL/yyypcom, +INCL/yytblc`

AUTHOR
Theresa Breckon
NAME
Address - address file for mail system

SYNOPSIS

DESCRIPTION
Address is the user address database for use with the mail system. This file contains information on each valid user of the system. Although many systems support the notion of an user authorization file, the access to this file is typically limited to users possessing hoarded privileges. In addition, the format of the file will prevent user programs from being written in a portable fashion. By providing an utility ('addr') to convert the contents of the user authorization file to a standard form in a file accessible to all, the above discrepancies are alleviated.

IMPLEMENTATION

Each line of the file is formatted as blank separated tokens, with the following positional meanings:

username  home-directory  system-info*  "descriptive info"

where 'username' is the user's login name, 'home-directory' is the user's home directory, 'system-info*' is 0 or more columns of system specific information, and "descriptive info" is auxiliary information about the user. Typically, all of this information is obtained by 'addr' from the system authorization file. The only required information is contained in the first two columns; it is absolutely necessary that the home-directory string represent an absolute directory name for the mail system to work correctly. This file should be kept in sorted order.

FILES

EXAMPLES
The following is an excerpt from the RTSGVX system:

belew disk$user1:[belew] [330,005] "don belew"
calico disk$user1:[calico] [311,002] "kate gilpin"
dj disk$user1:[dj] [313,001] "don rondeau"
gp disk$user1:[gp] [311,003] "gloria petit"
lapope disk$user1:[lapope] [330,002] "lev pope"
rab disk$user1:[rab] [330,004] "bob belshe"
rps disk$user1:[rps] [226,006] "r. p. singh"

A complete, absolute home directory spec for this system requires the specification of a device and an UIC. The third column is the UIC of the user, and is used by other primitive functions. The last column contains the first and last names of the user as given when the account was created.
SEE ALSO
addr(tool)

AUTHOR
Joe Sventek. Modified by Wei-Ling Chang.
NAME
ascii - the American Standard Code for Information Interchange

SYNOPSIS
See below.

DESCRIPTION

<table>
<thead>
<tr>
<th>decimal</th>
<th>hex</th>
<th>character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>@ (NL)</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>A (SH)</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>B (STX)</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>C (ETX)</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>D (ETB)</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>E (ENQ)</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>F (ACK)</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>G (BEL)</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>H (BS)</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>I (HT)</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>J (LF)</td>
</tr>
<tr>
<td>11</td>
<td>0B</td>
<td>K (VT)</td>
</tr>
<tr>
<td>12</td>
<td>0C</td>
<td>L (FF)</td>
</tr>
<tr>
<td>13</td>
<td>0D</td>
<td>M (CR)</td>
</tr>
<tr>
<td>14</td>
<td>0E</td>
<td>N (SO)</td>
</tr>
<tr>
<td>15</td>
<td>0F</td>
<td>O (SI)</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>P (DLE)</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>Q (DC1)</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>R (DC2)</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>S (DC3)</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>T (DC4)</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>U (NAK)</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>V (SN)</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>W (ETB)</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>X (CAN)</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>Y (EM)</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>Z (SUB)</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>[ (ESC)</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>\ (ES)</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>] (GS)</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>^ (RS)</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>_ (US)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>decimal</th>
<th>hex</th>
<th>character</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>20</td>
<td>(space)</td>
</tr>
<tr>
<td>33</td>
<td>21</td>
<td>!</td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>&quot;</td>
</tr>
<tr>
<td>35</td>
<td>23</td>
<td>#</td>
</tr>
<tr>
<td>36</td>
<td>24</td>
<td>$</td>
</tr>
<tr>
<td>37</td>
<td>25</td>
<td>%</td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>&amp;</td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>'</td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>(</td>
</tr>
<tr>
<td>41</td>
<td>29</td>
<td>)</td>
</tr>
<tr>
<td>42</td>
<td>2A</td>
<td>*</td>
</tr>
<tr>
<td>43</td>
<td>2B</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>2C</td>
<td>,</td>
</tr>
<tr>
<td>45</td>
<td>2D</td>
<td>-</td>
</tr>
<tr>
<td>46</td>
<td>2E</td>
<td>.</td>
</tr>
<tr>
<td>47</td>
<td>2F</td>
<td>/</td>
</tr>
<tr>
<td>48</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>49</td>
<td>31</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>51</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td>53</td>
<td>35</td>
<td>5</td>
</tr>
<tr>
<td>54</td>
<td>36</td>
<td>6</td>
</tr>
<tr>
<td>55</td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td>56</td>
<td>38</td>
<td>8</td>
</tr>
<tr>
<td>57</td>
<td>39</td>
<td>9</td>
</tr>
<tr>
<td>58</td>
<td>3A</td>
<td>:</td>
</tr>
<tr>
<td>59</td>
<td>3B</td>
<td>;</td>
</tr>
<tr>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
</tr>
<tr>
<td>61</td>
<td>3D</td>
<td>=</td>
</tr>
<tr>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
</tr>
<tr>
<td>63</td>
<td>3F</td>
<td>?</td>
</tr>
<tr>
<td>Decimal</td>
<td>Hex</td>
<td>Octal</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>@</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>66</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>67</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>68</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>69</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>70</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>71</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>72</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>73</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>74</td>
<td>4A</td>
<td>J</td>
</tr>
<tr>
<td>75</td>
<td>4B</td>
<td>K</td>
</tr>
<tr>
<td>76</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td>77</td>
<td>4D</td>
<td>M</td>
</tr>
<tr>
<td>78</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>79</td>
<td>4F</td>
<td>O</td>
</tr>
<tr>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>81</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>82</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>83</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>84</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>85</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>86</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>87</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>88</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>89</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>90</td>
<td>5A</td>
<td>Z</td>
</tr>
<tr>
<td>91</td>
<td>5B</td>
<td>[</td>
</tr>
<tr>
<td>92</td>
<td>5C</td>
<td>\</td>
</tr>
<tr>
<td>93</td>
<td>5D</td>
<td>]</td>
</tr>
<tr>
<td>94</td>
<td>5E</td>
<td>~</td>
</tr>
<tr>
<td>95</td>
<td>5F</td>
<td>_</td>
</tr>
</tbody>
</table>
NAME

environ - user environment

SYNOPSIS

see below

DESCRIPTION

A set of environment variables and their values is made available by the shell when a process begins. The tool printenv and the builtin shell commands setenv, unsetenv, and resetenv are available to manipulate the values of the environment variables. The functions envset, envget, and envrmv can be used in a program to access environment variables (see envlib(lib)). Below is a list of environment variables (and a description of their values) which are currently used by various tools (see /etc/env/login.rc for the default values of these variables):

PATH - A list of directories that the shell and any tool that spawns to another tool use to search for a command.

INCL - A list of directories that rat4 and includ use to search for include files.

MAN - A list of directories that man uses to search for manual entries.

LIB - A list of directories that the link tool, ld, uses to search for specified libraries.

TERM - The terminal type of the user, used by the vi tool on the VAX.

SEE ALSO

sh(tool), setenv(tool), unsetenv(tool), printenv(tool), resetenv(tool), envlib(lib)

AUTHOR

Theresa Breckon
NAME
Hnames - rewriting rules for mail addresses

SYNOPSIS

DESCRIPTION
It is often necessary to provide extremely hybridized addresses to force a message to be gatewayed from the system upon which it was composed to the system upon which the intended recipient resides. Eventually, all systems will support a common addressing syntax; until that time, it will be necessary for the user to present hybridized addresses to the mail system. The rewriting rules embodied in 'hnames' permit the user to use standard addresses (user @ host); the system then produces the hybridized address for the user. Each line of the file consists of two tokens separated by blanks, with the following positional meanings:

host-string replacement-pattern

'host-string' is of the form "@hostname", while 'replacement-pattern' is any sequence of characters up to the terminating newline, and MUST contain the character ' & '. The rewriting rules are used as follows:

repeat {
    get next address
    break into form left-side@host
    lookup @host in rewriting rules
    if (! found)
        break
    replace all occurrences of ' & ' in replacement
    text by left-side
    push replacement text back onto input
}

The standard composition programs provided by the system load two files containing rewriting rules in the following order: ~/msg/hnames, then ~/hnames. The first file loaded contains system-specific rewriting rules available to all users of the system, while the second file is the user's own set of rewriting rules kept in his home directory. By loading the files in this order, any conflicts in host name give the user's definition preference.
EXAMPLE

Consider the following entries in \texttt{msg/hnames}:

\begin{verbatim}
@lbl-unix  &@lbl-unix@lblg
@berkeley &@berkeley@lbl-unix
@ucbvax    ucbvax!&@lbl-unix
@decvax    decvax!&@ucbvax
\end{verbatim}

The following name translations will be performed

\begin{verbatim}
user@lblg  ==> user@lblg
user@lbl-unix ==> user@lbl-unix@lblg
user@berkeley ==> user@berkeley@lbl-unix
        ==> user@berkeley@lbl-unix@lblg
user@ucbvax ==> ucbvax!user@berkeley
        ==> ucbvax!user@berkeley@lbl-unix
        ==> ucbvax!user@berkeley@lbl-unix@lblg
user@decvax ==> decvax!user@ucbvax
        ==> decvax!decvax!user@berkeley
        ==> ucbvax!decvax!user@berkeley@lbl-unix
        ==> ucbvax!decvax!user@berkeley@lbl-unix@lblg
\end{verbatim}

FILES

\texttt{~/.hnames}, \texttt{~/.msg/hnames}

AUTHOR

Joe Sventek
NAME
Hosts - Host routing table

SYNOPSIS

DESCRIPTION
Once the mail delivery system has received a message to deliver to an user on another host, it must figure out how to get the message to its destination. Unlike other systems which decide which network to use dependent upon various pieces of fruit salad in the address, the tools delivery system requires that the binding of network protocol to host name be provided in this file, ~msg/hosts. As networks become more commonplace, it will not be unusual for there to exist multiple routes over different networks between two hosts. As such, it is necessary that the delivery system support the capability to adaptively route the message, dependent upon the availability of the various connecting network substrates. If more than one route exists between hosts, the order of trial for the different networks is encoded in this file, also. Each line of the file '~msg/hosts' consists of three blank-separated fields, with the following positional meanings:

dest-host  route-host  protocol-to-use

where ‘dest-host’ is the host we are trying to reach (no ‘@’ here), ‘route-host’ is the actual host to send the message to (again, no ‘@’), and ‘protocol-to-use’ describes the protocol to use to send the message to ‘route-host’.

If ‘route-host’ is the character ‘*’, this is shorthand to mean that ‘route-host’ is identical to ‘dest-host’. If ‘protocol-to-use’ is of the form

|"command to execute"

The command will be spawned, with the message file fed to the command as its standard input. In this way, additional network support can be achieved without rebuilding the delivery system.

If the last character before the newline is an atsign (‘@’), this implies that the next line is an alternate route to the same destination. The alternate routes are tried in the order listed in the file until the message is successfully delivered.

If a particular host cannot be found in the file, and an entry for the host ‘default’ exists, then the action specified will be taken. In this situation, an asterisk (‘*’) in column two indicates that ‘route-host’ is identical to the host we were unable to find in the table.

EXAMPLE
Suppose the following host table
Then the following routing decisions will be made:

* user@lblg - first try decnet; if that fails, try hyperchannel
* user@lblh - only try decnet
* user@unix - delivr will spawn msgtouucp with its standard input receiving the SMTP-encoded temporary file. It is msgtouucp's responsibility to deliver the mail, and to exit with an error if it was unable to do so. In this way, delivr can requeue the message for later delivery.
* user@foo - hot potato route the message to lblg, which will have the resources to deliver the message.

If the default line had been of the form

default * decnet

then delivr would have attempted to deliver the mail to 'foo' directly via decnet.

FILES

~msg/hosts

SEE ALSO

hnames(file)

DIAGNOSTICS

If a host could not be found in the file and there is no default action, the message is returned to the sender with an error indicating that they have specified an invalid host.

AUTHOR

Joe Sventek. Modified by Wei-Ling Chang.
NAME

Lnames - Binding of host names to networks

SYNOPSIS

DESCRIPTION

One of the unfortunate facts of existing systems is that a host can be known by different names on different networks. Hopefully, this situation will change over time; until that great day, it is necessary for the software to know the names by which a particular host is known on each network to which it is directly connected. The above binding is described in the file `~msg/lnames`. Each line of this file consists of two blank-separated tokens, with the following positional meanings:

```
network-name @host-name
```

The value for network-name are the same used in the file `~msg/hosts`. In addition, a value for the network-name "default" should be placed here. This value will be used when composing the mail, as well as, whenever 'gthost' is invoked for an unknown network.

EXAMPLE

The following is the contents of the `rtsgvx` `~msg/lnames` file:

```
default @rtsgvx
```

FILES

```
~msg/lnames
```

SEE ALSO

```
hosts(file)
```

AUTHOR

NAME
Malias - file of mail aliases

SYNOPSIS

DESCRIPTION
A critically important feature of any mail system is the ability to define alias names which expand into one or more addresses at mail delivery time. Besides permitting groups to be identified with a single name, the feature also permits mail forwarding, network teleconferencing and many other useful applications. The tools mail system provides for a system-wide alias file, the contents of which are valid for local and remote users alike. In addition, when using the standard composition utilities, a user-specific alias file (in the user's home directory) is loaded for use by the user. In this latter case, the aliases are loaded in the order described; in this way, conflicts in the alias name space are resolved in favor of the most recently loaded value for the alias.

The expansion of aliases is similar to the expansion of macros in programming languages. The recursive nature of the expansion (and the possibility for infinite recursion if the aliases are not well-formed) requires that some care be taken when creating and modifying a specific alias definition file. Each of the alias files consists of one or more alias definitions. Each alias definition is of the following form:

alias-name: [address] [,address]* ;

The 'alias-name' should consist of alphanumerics, together with the characters minus '-' and underscore '_'. Each 'address' can be any valid user address, including another alias name. Commas ',' must separate addresses; an alias definition can be continued to another line by simply breaking the definition at one of the commas. The end of an alias is indicated by the appearance of a semi-colon ';'.

If it is desired to prevent the expansion of an alias name during mail delivery, surrounding the name with quotes "" will do the trick (see examples below).

Two special forms of the alias definition are permitted in the system alias file:

* The alias-name can be separated from the definition by a question mark '?' instead of a colon ':'. Aliases so flagged will NOT be listed by the 'users' utility.
* A valid form for the replacement part of the definition is

"command to execute";

This form will cause the message to be delivered as the standard input to
a process running the specified command.

EXAMPLE
Suppose the following definitions in the file `~msg/malias`:

```
csg(central support group): chang, jane, leres, tab, 
    van, vern, marshall, jpl, rick, sam, barale; 
rtsg: rtsg@lblg; 
rmtxeq: |"rmtxeq ??~mail/rmtxeq.log"; 
msg(msg mail system)? "msg", chang; 
netnpr? ;
```

Perusal of these aliases displays all of the major features of possible alias definitions:

1. Multi-line definitions are achieved by simply breaking the definition at a comma (i.e: csg ).

2. Comments about the alias name are achieved by placing the comments inside of parentheses next to the alias name before the ':' of '?' separator (csg & msg).

3. Network addresses can be used in the replacement part of the definition (rtsg).

4. A command to spawn can be the replacement part of an alias in the system alias file (rmtxeq).

5. A valid address can be defined as an alias also (the name spaces are separate). In this situation, it is important to be sure to place the address inside of quotes in the definition part to avoid infinite recursion (msg).

6. For those addresses for which mail is not desired, they can be aliased to nothing (netnpr).

FILES
`~msg/malias, ~/malias`

SEE ALSO
stmail(tool), users(tool)

AUTHOR
NAME
    Msgslp - help file for msg utility

SYNOPSIS

DESCRIPTION
    This file provides the help information used by the 'msg' utility. The only entry in the archive which should ever be changed is the news module; this should be changed to reflect any changes in the workings of the mail system. The file is an archive as maintained by 'ar', and has to reside in the /msg directory. To change the news entry in the archive, one must perform the following steps:

    % ar -xv /msg/msgslp news
    % ed news
    <make your modifications and write the file>
    % ar -uv /msg/msgslp news

Any other form of modification (especially editing the archive file itself) can have dire consequences in the workings of 'msg'.

FILES
    /msg/msgslp

SEE ALSO
    msg(tool), ar(tool)

AUTHOR
NAME

Sufile - list of mail system super-users

SYNOPSIS

DESCRIPTION

Several system-management utilities are included with the system to permit control over various aspects of the delivery system. The user whose name is listed in "~msg/sufile" has the permission to run those system-management utilities.

EXAMPLE

The contents of the author's ~msg/sufile are displayed below:

    sysmgr
    system

FILES

~msg/sufile

SEE ALSO

mstop(tool), mtrace(tool), nalias(tool), netdlv(tool)

AUTHOR

Joe Sventek. Modified by Wei-Ling Chang.
NAME
Addr - generate address file entries

SYNOPSIS
addr

DESCRIPTION
Addr generates the address file entries needed by the software tools mail system. ‘addr’ is an operating-system-specific utility which reads the operating system’s user authorization file and generates the standard entries for the address file. It is assumed that ‘addr’ must be run from a privileged account in order to access the user authorization file.

The output from ‘addr’ is not guaranteed in any particular order; as a result, it is most often piped into sort before writing the address file, as in

```
addr | sort >~msg/address
```

FILES
system-specific authorization file

SEE ALSO
address(file)

DIAGNOSTICS
An error message is displayed if the authorization file could not be opened at READ access.

AUTHOR
Originally written by Joe Sventek. Modified by Whei-Ling Chang.
NAME
backups - How to do backups

SYNOPSIS
Shutdown system; Use (standalone) DSC2

DESCRIPTION

1. Login with Username = SHUTDOWN, Password = NONE (type the word 'NONE')

2. Answer the question

   HOW MANY MINUTES BEFORE SHUTDOWN?

   If no one is using the system you can specify 0, otherwise give them enough time to perform a cleanup (5-20 minutes)

3. When SHUTDOWN completes with the message,

   SYSTEM SHUTDOWN COMPLETE-USE CONSOLE TO HALT SYSTEM

   Use the operator console to perform the following steps:
   Hit CTRL-P (Key on front panel must be in "LOCAL" position)

   Response is:

   >>>

   Type HALT <Ret>

   Response is:

   HALTED AT.....

   Type BOOT DSC <Ret>

4. The computer will pause for a few seconds while it reads the console floppy. Then it types:

   Please insert first system diskette or cartridge
   and type <Ret>

5. Now open the door to the VAX, pull open the floppy disk drive from inside the cabinet and remove the console floppy from the drive. Get the first floppy for the DSC program from the Wright Line cabinet (the title of this floppy is "STAND/ALONE11780 DSC2 FLP1") and put it in the drive.
Then hit <Ret> and wait for next messages.

6. After approximately six minutes, it will type

   VAX/VMS Version V2.0 6-APR-80 16:55

   Please insert second system diskette or cartridge
   and type <Ret>

7. Now replace the floppy in the drive with the second floppy for the DSC program titled "STAND/ALONE 11780 DSC2 FLP2" and then hit <Ret>.

8. After the program finishes loading (about 1 minute later) the prompt will appear on the terminal.

   DSC2>

9. A backup is performed by copying the contents of a disk onto another "blank" disk (called an "output pack"). To choose the proper output pack for your backup, use the oldest backup pack for the drive you are backing up. See label(maint) for a description on how to determine the oldest pack for a particular device. Set the disk drives not to be written upon to READ ONLY, insert the output pack in one of the drives (RM03, if input is RM03; RM05 if is RM05) and then give the command line to DSC. The command line is in the form:

   output device (optional verify switch)=input device

   (e.g. DRnn:/VE=DRnn:)

   (The RM03 drives are DRA0: and DRA1;
   the RM05 drives are DRC0: and DRC1:) example:

   DRA1:/VE=DRA0:

   If any error messages, check the VAX Utilities Reference Manual, under DSC chapter. (Chapter 4)

10. DSC will perform a copy pass, followed by a verify pass. The full DSC will therefore take approximately 50 minutes for an average RM03, and ? minutes for an average RM05. When DSC is finished, the prompt

   DSC2>

   will be displayed.
11. If no errors and more backups are to be done, go back to step 9.

12. If no errors and no more backups are left, halt the machine by saying:

   CTRL -P
   >>>
   HALT <Ret>
   HALTED AT.....

13. Now follow these steps:

   1. Replace the console floppy in the drive

   2. Label the packs you used

   3. Return the DSC floppy disks to the Write Line cabinet,

   4. Put the ORIGINAL packs in the cabinet, replacing the current packs in the drives with the NEW packs just copied

   5. Turn off READ-ONLY switches

   Reset system with VMS.

   >>>
   BOOT

SEE ALSO
VAX Utilities Reference Manual, pp. 4-1 thru 4-33, chapter 4.

AUTHOR
Sheldon Wong and Annette Michael
NAME
   cpuaf - copy a sysuaf file, removing passwords

SYNOPSIS
   cpuaf inuaf outuaf

DESCRIPTION
   CPUAF creates a duplicate of the input uaf file that has the password field destroyed. This program is usually run at system boot up time and is used to prevent users from attempting to decrypt other users passwords (as if this process is trivial!)

SEE ALSO
   cp(tool)

AUTHOR
   Craig Leres
NAME

install - perform a system installation

SYNOPSIS

install prog fromdir [-s] [-p] [-m section]
    [-i name] [-o file] ...

DESCRIPTION

Install does the file moving portion of a system installation (i.e., moves the source, procedure, manual, etc., files from a user's directory to a system or project master directory). Install should only be run by the appropriate system or project installer and only when the installer is logged in as the project's "super..." name.

The arguments to install are taken directly from the standard RTSG change form:

prog is the name in the "Program" box.

fromdir is the name in the "from:" box (the user's root directory name).

-s appears if the "src" line is checked.

-p appears if the "proc" line is checked.

-m appears if the "man" line is checked. The section name that follows the "man" line should be put after the "-m" flag (e.g., "-m tool" if the section name next to the "man" line on the form was "tool").

-i appears if the "incl" line on the form is checked. The name that follows the "incl" on the form should go after the "-i" flag (e.g., "-i cmacro" if the name following the "incl" line was "cmacro"). If there is more than one name on the incl line, a "-i" flag should be used for each.

-o appears if the "other" line on the form is checked. The name that follows the "other" on the form should go after the "-o" flag (e.g., "-o tf/rtdb/xwav" if the name following the "other" line was "tf/rtdb/xwav"). If there is more than one name on the "other" line, a "-o" flag should be used for each.

Install first checks that each of the files to be moved actually exists. If any of the files that the user wants moved doesn't exist or can't be read, the message

-cant find xxxxx

(where "xxxxxx" is the name of the file that's missing) is output to your terminal and nothing will be installed. If all of the files exist, they will be copied to
the appropriate place in the master directory. As each one is copied, the
message

 -installing xxxxx

is output to your terminal. After all of the files have been installed, install
sends mail to the user telling the user which files were just installed and to
unlock and remove all local copies.

FILES

 ~/ins.tmp for sh commands to copy files /usr/manager/insdir.d for
macro input to build sh commands

SEE ALSO

DIAGNOSTICS

AUTHOR

Van Jacobson

BUGS/DEFICIENCIES

A maximum of 29 arguments may be given to install. (the limit can be raised
by adding additional 'i_addarg' calls to the install script)
NAME
label - how to label disk packs

SYNOPSIS
Put 4 labels on each disk pack. One when the pack is initialized. One for the serial number. One for the machine type and data type. One for the log of backups made.

DESCRIPTION

Figure 1. Typical disk with labels. (side front view)

```
[Diagram of a disk with labels marked 1, 2, 3, 4]
```

Figure 2. Top view of disk.

```
[Diagram of a top view of a disk with label 1]
```

For the RTSG VAX disk packs, there are usually four labels on each container of the disk.

1. The label on the top cover has initialization information including the method of initialization (BAD or PACKINIT), the number of bad blocks, the initials of the person who did the initialization, the date of the initialization, the volume label of the pack (if any).
2. The leftmost label on the side of the cover has BACKUP information including the name of backup utility used (DSC or BACKUP), the type of usage (is pack source or destination), the initials of person who did backup, the date of backup, the pack number of the source pack if this pack is a destination device, or the pack number of the destination pack if this pack is a source device.

3. The center label on the side of the cover has the SERIAL NUMBER of the pack, which comes from the serial number assigned when a pack is purchased. This number is also written with a permanent marker on the top disk surface platter.

4. The rightmost label on the side of the cover has the CONTENTS information including the machine (RTSG VAX 11/780), the type of disk (SYSTEM or USER PACK), the device used (RM03 #0 or #1 or RM05 #0 or #1).

Examples of labels

1. Top label - information is written here only once at initialization.

```
PACKINIT/VERIFY 0 BAD BLOCKS
33 30-SEP-81
LABEL: VAXMSR10
```

2. Leftmost label - information is added here each time the pack is used in DSC or BACKUP.

```
DSC Destination; SW, 25SEP81(from #03)
DSC Source; SW, 10OCT81(to #29)
```

3. Center label - only the serial number of the pack is placed here at the time the pack is purchased.

```
244
```
4. Rightmost label - information is added here the first time the pack is used.

FILES
None

SEE ALSO
VAX Utilities Reference Manual, Ch. 6.

DIAGNOSTICS
A message is printed if a file cannot be opened.

AUTHOR
Sheldon Wong

BUGS/DEFICIENCIES
None
NAME
   lbkwic - make library kwic index

SYNOPSIS
   lbkwic [any argument]

DESCRIPTION
   lbkwic produces a kwic index of the library routines in ./src. It uses the file
   ./lib/xref.lis, created by previous use of the script lbxref. If xref.lis doesn't
   exist, then lbxref is called to produce it.

   The resulting permuted, processed, and sorted cross-reference is written
   onto the file ./lib/kxref.lis. In addition, if any non-null argument is specified
   (see synopsis), then the file is formatted and sent to the line printer.

   Note that since the kwic index is written onto a file in the current directory,
   the user must have write permission in the current directory. Furthermore,
   the libraries should be in standard software tools format, for the first cross-
   reference to work. See the manual entry for lbxref for a description of the
   required format.

FILES
   ./lib/kxref.lis - where the kwic index listing is written

SEE ALSO
   lbxref (maint) kwic (tool)
NAME
mkform - output numbered forms to standard out

SYNOPSIS
mkform [-f first] [-n copies] [file]

DESCRIPTION
Mkform is a tool used to produce numbered forms (like Change forms or SPR forms). It reads the prototype form from "file" (if no file is given, the form is read from standard in). The "-f" flag sets the first form number (the default is "1"). The "-n" flag sets the number of forms to output (the default is 100).

The prototype form is output unchanged except for the following escape sequences:

is turned into the current form number.

is turned into a formfeed.

is turned into a carriage return (used to overprint a line).

is discarded (a 'hidden newline').

Any other character following an "@" is replaced by that character. E.g., "@@" becomes "@".

FILES
None.

SEE ALSO
lnpr(tool)

AUTHOR
Van Jacobson

BUGS/DEFICIENCIES
Because of the ridiculous value of MAXLINE (80), forms with more than 80 character lines are very difficult to deal with (this is a limitation of "ed", "ch" and "pr", not "mkform").
NAME
Mstop - cause the mail system to gracefully close down operations

SYNOPSIS
mstop

DESCRIPTION
Mstop sends a special message to the mail delivery system to cause it to close down gracefully. Upon invocation, 'mstop' compares the username of the user with a list of user's who have been accorded "super-user" privilleges for the mail system. If the username is found in the list, the special message is sent to the delivery system, causing the system to gracefully close down.

FILES
~msg/sufile

SEE ALSO
sufile(file)

DIAGNOSTICS
Only those users in the "~msg/sufile" have the "super-user" privilege to run this program.

AUTHOR
Joe Sventek. Modified by Wei-Ling Chang.

BUGS/DEFICIENCIES

-1-
NAME
Mtrace - change the logging level for delivery system traces

SYNOPSIS
mtrace [level]

DESCRIPTION
Mtrace sends a special message to the mail delivery system to cause the
trace message level to be modified for MAILER and DELIVR. Upon invocation,
'mtrace' compares the user name with a list of user's who have been
accorded "super-user" privileges for the mail system. If the name has been
found in the list, the special message is sent to the delivery system, causing
the changes of logging level.

Valid trace level arguments and their meanings are:

0 No error messages are entered in log files (you get this if you omit the
   'level' argument)
1 Log only errors internal to the workings of each process (eg. a file created by
   one subroutine was not readable by another subroutine in the same process)
2 Log communication errors between processes
3 Log information necessary to trace the flow of each message through the
   delivery system
4 Log auxiliary information extraneous to message flow.

Each level includes those below it (4 => 1-4). Each message in the log files is pre­
ceded by the level to which the message corresponds.

FILES
~msg/sufile

SEE ALSO
sufile(file)

DIAGNOSTICS
Only those users are listed in the "~msg/sufile" have the privilege to run this
program.

AUTHOR
NAME
nalias - cause mail system to reload address and alias files

SYNOPSIS
nalias

DESCRIPTION
Nalias sends a special message to the mail delivery system to cause it to
reload the files '~msg/address' and '~msg/malias'. Upon invocation, 'nalias'
compares the name of the user with a list of user's who have been accorded
"super-user" privileges for the mail system. If the username is found in the
list, the special message is sent to the delivery system, and after the files are
reloaded, the user is notified that the requested action has been performed.

FILES
~msg/address, ~msg/malias, ~msg/sufile

SEE ALSO
address(file), malias(file), sufie(file)

DIAGNOSTICS
Only those users listed on the "~msg/sufie" have the "super-user" privilege
to run this program.

AUTHOR

BUGS/DEFICIENCIES
NAME
Netdlv - enable/disable network delivery

SYNOPSIS
netdlv {on|off}

DESCRIPTION
Netdlv sends a special message to the mail delivery system to cause network
delivery of messages to be enabled/disabled. Upon invocation, 'mtrace' com­
pares the user name with a list of user's who have been accorded "super­
user" privileges for the mail system. If the username is found in the list, the
special message is sent to the delivery system, causing the network deliver
to be enabled/disabled.

FILES
~msg/sufile

SEE ALSO
sufile(file)

DIAGNOSTICS
Only those users listed on the "~msg/sufile" have the privilege to run this
program.

AUTHOR
Joe Sventek. Modified by Wei-Ling Chang

BUGS/DEFICIENCIES
NAME

portmod - software tools port from vax to modcomp

SYNOPSIS

(see description)

DESCRIPTION

1. Creating Port Tape

1.1. Login on Vax

Make sure that the tape drive 'mta0' is available. If so, login on the vax from a hard copy terminal. Otherwise, wait until drive 'mta0' is available. Do not allocate the tape drive. It will be allocated by the script.

1.2. Execute 'portemall' Script

The portemall script assumes that your current working directory contains a src and incl sub-directory, and your include search path file (~/incl.sp) contains "./incl" as the first line. It is safest to have a separate "port" directory with "src" and "incl" sub-directories. You also need a port search path file (~/port.sp). The normal content of the port search path file is a single line "/usr/manager/port/dev1port". Change directory to "port". Type

% ~/.%port/portemall

This will generate the tape to port all of the portable files to the modcomp.

Portemall automatically generates a rat4 boot for the modcomp. This is why you need the src and incl sub-directories in your port directory. If for some reason you do NOT wish to generate the rat4 boot, type: "~/.%port/portemall -c". Normally, you would want to generate the rat4 boot for the modcomp.

1.3. Watch and Make sure Files are Copied Correctly

Portemall will generate a short description on the terminal of the files being written to the tape. This is why you should be logged in on a hard-copy terminal. Keep the description for record purposes. Make sure that portemall was able to copy all files. If there were any error messages, you must correct the problem and start over again. You can not use the tape as it exists, as files on the modcomp would be put in the wrong place. If there was an error, and if the script didn't abort (because wmodt doesn't abort if it can't copy a file), you MUST correct the problem and begin again.
2. Reading Port Tape

2.1. Login on Modcomp

Now you should have the tape to do the port to the modcomp. Logout on the vax, dismount the tape and label it as the "VAX_TO_MODCOMP PORT TAPE" with the date and your name. Go to the Modcomp/DEV1. Mount the tape on the modcomp (on mt). Login on the modcomp from a hard-copy terminal so that you will have a permanent record of the port.

2.2. Execute 'rdvaxpt' Script

The next step of the port assumes that your current working directory is called port, and that all of your search paths have "/" as the first path. For example, .%/incl would have "/incl" as it's first line. You must also have write permission in "port". Type

```
% rvaxt rdvaxpt
% rdvaxpt
```

2.3. Generating Rat4 Boot and Reading Files

Follow any directions, and answer any questions rdvaxpt asks you. Rdvaxpt will first attempt to remove and recreate all of the appropriate port sub-directories (bin, incl, lib, man, manager, proc, src, test, etc, misc). If you do not wish it to remove and recreate these directories you can type "rdvaxpt -n". However, it is strongly recommended that the removal and recreation be done to protect the validity of the port. Rdvaxpt then generates the rat4 boot. If you do not wish to generate the boot, type "rdvaxpt -c". Again, it is strongly recommended that you build the boot. (To combine the flags, type "rdvaxpt -nc".)

Rdvaxpt will then read the rest of the port files off of the tape into the appropriate sub-directories. Information will be printed about each file as it is read off of the tape.

2.4. Watch and make sure files are read Correctly

If there are any error messages about not being able to read a file, rdvaxpt should be aborted. Otherwise the rest of the files could be read off of the tape in the wrong order. The problem must be resolved before continuing. This may involve re-writing the port tape on the VAX.
2.5. Compiling Ported Files

After reading all of the files from the tape, rdvaxpt will then recompile all the sources for the portable programs. It will take many hours. Keep track of all compilations that fail. These failures mean there are portability problems with the code. Someone who is qualified should fix the problem on the Vax and have the code reported.

2.6. Completing the port tape

When rdvaxpt has finished all of the compilations, it will print a message informing you the port has been completed. You should then remove the tape and store it in the proper place.

2.7. Testing Tools

The next step is to test all of the tools in port/bin. Any tools that do not work should be investigated. The tool should not be moved into /usr until the problem has been resolved on the vax. The fixed tool should then be reported to the modcomp and re-tested.

3. Moving the Port into /usr

The next step is to move all of the tested tools, libraries, manuals, etc into /usr. DO NOT move buggy tools or libraries into /usr. Resolve all port problems before doing this move. If everything has been tested, do the following (or have the appropriate person do the following):

1. Login as superuser
2. Change directory to the "port" directory
3. Execute the script 'mvvaxpt' to move the result of the port into /usr and /etc:

   % mvvaxpt

FILES

port/oldport - record of files written to port tape (on vax).

On the VAX, in the directory /usr/manager/port/dev1port:

   portlist - list of portable files
   portemall - script to create port tape on the VAX.
   rdvaxpt.all - script to read the port tape onto the MODCOMP.
   mvvaxpt.all - script to move tested, ported files into /usr and /etc on the MODCOMP.
NAME
prtallvax - port all tools to a foreign vax

SYNOPSIS
(see description)

DESCRIPTION
Prtallvax will port the software tools to a foreign vax. It assumes the foreign vax is running a version of vms with the backup utility. (Note that the backup utility comes with version 2.3 or higher of VMS.)

Make sure that tape drive mta0 is free. Log in as system manager on the vax you are porting tools from. Use a hardcopy terminal and keep the output. Type

```
% prtallvax
```

Answer the questions prtallvax asks. You will have to know the names of the disk drives which /usr, /etc, and /misc are on. Give the name of the drive in the form prtallvax specifies: no trailing colon is allowed. When it asks to mount the tape, mount the tape, and type "y".

When 'prtallvax' is finished, you have a tape which can be used to port the tools to a foreign vax. Take the tape to the foreign vax, log in as system manager, again, on a hardcopy terminal, and type:

```
$ allocate mta0
$ mount/foreign mta0 tools tape
$ backup tape:install. []
$@toolbuild.com
$! note and follow any directions toolbuild.com gives to you
$! when it's finished, you now have the tools
$ rm toolbuild.com
```

You can replace 'mta0' in the preceeding commands by the name of any other tape drive. Toolbuild.com will ask you several questions. It will ask you which disks /etc, /misc, /mnt, /tmp, and /usr should go on, what the uic's of the directories /etc, /misc, /tmp, and /usr should be, and what the network name of your vax is (or will be). Answer these questions in the form prtallvax specifies.

At this point the tools sources will be installed. There are still several things you have to do to make the tools usable. These are explained toolbuild.com.
AUTHOR
   Todd Hammond
NAME
railroad - perform a quick system installation

SYNOPSIS
"Oh, gosh, I don't have time to wade through all those forms!"

DESCRIPTION
Railroading is a time-honored RTSG practice in which built-in system "safety" features such as Change Forms are by-passed in order to enhance system modification speed. "AMHACKER" privilege must be set.

To execute "railroad", do the following steps:

1. if you haven't already tested your fix, don't start now
2. if someone's watching, fill out an RTSG Change form. Grab Bob or someone of his ilk to do a "Railroad Authorization" (see RailAuth(file)).
3. pick up the necessary sources and objects. Better bring along a copy of that 'library' object you have hacked out.
4. move them over. Compile if you have the time. If not, objects will have to suffice.
5. What the hell. Port it to the Modcomp.

"Railroading" can do grave damage to the intricate Tools structure. Only Railroad if you think you know what you're doing. If you had to read this manual entry, you don't know, and if you try, your name will be logged and mail sent to bobup.

FILES
~<username>/ if it worked in your space, you might as well assure that you'll be using the same files when stuff's installed.

SEE ALSO
Bob Upshaw. But only if he's in a good mood.

DIAGNOSTICS
"Craig, what's this new tool doing in /usr/bin?" - Fatal error. You have failed to cover your tracks sufficiently. Wait until night and try again.

AUTHOR
Craig Leres and Vern Paxson
BUGS/DEFICIENCIES

Once upon a blue moon the installation will fail to work. Backdate the version numbers of the newly installed sources and SPR it.
NAME
sysgen - How to Change Sysgen Parameters

SYNOPSIS
Edit parameter file params.dat @sys$update:autogen Copy autogen.par to calsys.par

DESCRIPTION
The sysgen file on the Vax is created using the DEC-supplied command procedure autogen, which provides appropriate parameters based on the system configuration and takes into account interrelationships between parameters when making its calculations.

If parameter values are supplied to autogen, it will use those values instead of calculating them. The file from which it takes parameter values is /sys/sysexe/params.dat (sys$system:params.dat from DCL). Its format is simply PARAMETER=VALUE, with one parameter per line. To add or change parameter values, edit this file.

To create the new sysgen, run the DCL command procedure:

@sys$update:autogen

This is generally run from the system account since it requires both read and write access to system files.

The autogen command procedure will both update the current sysgen parameters and write a new version of the parameter file /sys/sysexe/autogen.par (sys$system:autogen.par from DCL). Since the sysgen file traditionally used as the standard system is calsys.par, the new sysgen must be copied from autogen.par to calsys.par (also in the /sys/sysexe or sys$system directory).

The procedure will also create or extend the system swap, page, and dump files if it calculates larger sizes for these than the sizes of the existing files.

Most sysgen changes will not take effect until the system is rebooted. After making changes, it is always advisable to reboot the system to make sure that it will come up properly with the new sysgen.

FILES
sys$system:params.dat (parameter data file)
sys$update:autogen.com (command procedure)
sys$system:autogen.par (sysgen created by autogen)
sys$system:calsys.par (sysgen used to boot the system)
sys$system:vsimages.dat (list of installed images created by autogen)
sys$system:swapfile.sys (swapfile, extended if necessary)
sys$system:pagefile.sys (pagefile, extended if necessary)
sys$system:sysdump.dmp (dumpfile, extended if necessary)

SEE ALSO
VAX/VMS System Management and Operations Guide, Chapter 10, gives information on the sysgen parameters. The VMS program sys$system:sysgen has a HELP command which will give information about the sysgen parameters and sysgen commands. The SHOW command can be used to show the current, default, minimum and maximum values for each parameter.

DIAGNOSTICS
The autogen command procedure gives error messages if it cannot create or extend files. If it is unable to extend the swap, page, or dump file, the message will say that the file must be created manually and specify the required size. If autogen is unable to create one of its scratch files or cannot find params.dat, the procedure will fail.

AUTHOR
Jane Colman
NAME
sysinst - system installations

SYNOPSIS
(See below)

DESCRIPTION

This document describes the steps which should be followed by the system approver and installer in order to update system programs or files. Section 1 discusses the steps to be followed by the approver, and steps 2 through 4 are to be followed by the installer.

In order to avoid system destruction, please remember the golden rule: "When in doubt, reject!"

1. Approval of System Change Forms

Before any system installation can occur, it must be approved by the system change approver(s). These approvers must:

(1) Using locks, make sure all files being installed are locked by the programmer. If not, reject.

(2) If any forms request "save old library", save the appropriate library onto oldbin. Be sure that an old library is not "saved" more often than once for a batch of installations, and that the old library being saved has had a chance to be backed up at least once.

(3) If any forms request "save old binary", save the tool binary(s), observing the same precautions as stated above for libraries.

(4) Check comments field of the installation form for:
- common program name/usage (e.g. "FTB=BCPL file manager primitives")
- Changes performed (e.g. "Fixed bug with line numbering not working across page boundaries.")

2. Order the forms

(1) Operating system changes and tools changes should be done in two separate groups.

(2) Within each group, all forms that require prior changes should follow the prior changes.
(3) Within each group, all forms that modify libraries should precede forms that modify programs.

3. Doing the installation

For system changes, login as supersys; for tools changes, login as superusr. For each form, do the following steps:

3.1. Move the files

If the form is in standard form, do

```
% install <prog.x> /mnt/person -s -p ...
```

Otherwise, use the cp tool to move the files as necessary.

3.2. Check all source files

(1) Check for correct revision history.
(2) Check for correct $VERSION if source is for a BCPL library.
(3) Check for $GLOBALS 10 (or no $GLOBALS at all) if BCPL library.
(4) Check for correct version number if assembly language:

```
INT V:xxx
V:xxx EQU 0
```

(5) Check for correct global size (or no globals at all) if assembly language:

```
GLOBL. COM 11
```

3.3. Check any procedures

(1) Standard form: if $1 = -l, procedure should produce a listing and (if linked) a load map.

3.4. Check any manual entries

(1) Standard form
3.5. Run procedure

(1) If requested, do

% prog.p

or follow instructions given. If the procedure writes to a hardware-
protected partition, switch the disk protect switch to the "none" posi-
tion, and switch it back to the center position after the procedure has
been run.

4. Documenting the installation

(For steps 1, 3, and 4 see dochange(maint). )

(1) For each tools installation form, update /etc/helpstuf/change.m and
    reformat.

(2) For each system installation form, enter change information in the
    weekly report (see separate documentation).

(3) For every form whose installation will affect the users, update and refor-
    mat /etc/helpstuf/news.

(4) Update the message of the day, /etc/motd.

5. Notifying the programmer

For every form, send mail to the programmer specifying either:

(1) The form was rejected. Put the form in his/her box and explain why it
    was rejected.

(2) The installation was successful. Request that the programmer unlock
    relevant files, remove personal copies, and remove test libraries if they
    were used. Be sure to list all files which were installed, so the program-
    mer knows which ones to unlock and remove.

SEE ALSO

dochange(maint)
DIAGNOSTICS

Many types of errors can occur. Since some may be dangerous if ignored, you should either try to find the cause of the error (and fix it if it was an error on your part) or seek help. If help is not available, stop the installation and reject it, outlining the errors. Return the form back to the programmer for fixing.

AUTHOR

Jane Colman and Bob Upshaw

BUGS/DEFICIENCIES
NAME
verify - using VFY to verify a disk

SYNOPSIS
Shutdown System, Use VFY2 to verify a disk pack.

DESCRIPTION

1. If you are going to VERIFY a pack, make sure you take the VMS system
down and boot up a limited system (USE OUSER.PAR in SYSBOOT), so no
user activity will go on while you are doing the VERIFY.

EXAMPLE:
(Login to SHUTDOWN, Password = NONE,
and shutdown the system)
Make sure the key on the front panel is in the
"LOCAL" position and then hit "CTRL-P".
The computer responds with:
>>>  
You type in: 
HALT

The computer responds with:
HALTED AT ........
>>>  
Now you type:
@DB0GEN
The computer responds with:
SYSBOOT>

You type in: 
USE OUSER.PAR
The computer responds with:
SYSBOOT>

You type in: 
CONTINUE
If you accidently hit CTRL-P and don't want to HALT the system
or otherwise, enter

SET TERM PROG
Then the '>>>' prompt will no longer appear

2. Now the VMS system gets booted, but with only one terminal, the Dec­
writer console, in the machine room.

3. Using the operator console, login to the privileged account MAINT. Then
disable logins by typing

SET LOG/INT=0
When you are finished with the VERIFY you should give the hardcopy produced by the VERIFY on OPAO: to the system manager. In all VERIFY operations you are going to perform, you MUST NOT ABORT the program with CONTROL-Y or CONTROL-C!

4. For the disk you are going to VERIFY, check to see that it has a directory [001003] and that any files in there are deleted. If [001003] does not exist for that disk, CREATE it.

5. If this disk is NOT a SYSTEM disk, proceed. If it is a SYSTEM disk, go to step 10. In this example we will assume that your disk is DRZ8: If not DRZ8:, substitute the correct device name.

6. First run the validity check to see if the number of free blocks on the target disk match, according to the index file and according to the bitmap.

   \$ MCR VFY2 DRZ8:

   During this program run, error messages may be printed out reporting on files with errors. See the VAX Utilities Manual, Chapter 6, for a list of error messages. (Most errors can be ignored, but keep a hardcopy of the entire session and leave it with the system manager). The last error message will be followed by a summary line for the entire disk-

   SUMMARY: MULT=nn, FREE=nn, BAD=nn
   MULT is the number of multiple block allocations
   FREE is the number of blocks marked free
   that should have been allocated
   BAD is the number of bad retrieval
   pointers in the file header.

   The number of free blocks according to the index file and the number of free blocks according to the bitmap will be printed. If these two numbers are the same and if there are no error messages, then the VERIFY is complete. If the number of free blocks according to the index file is larger than the number of free blocks according to the bitmap, then there are lost blocks. Go to step 7. If the number of free blocks according to the index file is smaller than the number of free blocks according to the bitmap, then there may be multiply-allocated blocks. If you get the error message "MULTIPLY ALLOCATED BLOCKS ...", stop here and see Bob Upshaw or Sheldon Wong. If Bob or Sheldon can't be found, go on to step 17. If there are files that are labeled with "FILE IS MARKED FOR DELETE" then delete these files by using the PIP utility. Enter

   MCR PIP

   The PIP utility responds with the prompt:
PEC>

specify the file ID you obtained from the VFY output in response to the PIP> prompt. For example, if VFY said that FILE ID 999,88 was marked for delete, enter:

```
/Fl:999/88/DE
```

After deleting all these files labeled with "FILE IS MARKED FOR DELETE" then continue with the next step.

7. To find lost files and put them in the directory [001003], do the following:

```
$ MCR VFY2 DRZ8:/LO
```

DO NOT ABORT!

8. If there are no multiply-allocated blocks, recover lost blocks by doing the following:

```
$ MCR VFY2 DRZ8:/RE
```

DO NOT ABORT!

9. To recover files with the error message BLOCK IS MARKED FREE, do the following to allocate blocks that appear to be available but are actually allocated to a file.

```
$ MCR VFY2,DRxx:=DRZ8:/UPD
```

where DRxx is the device name of a scratch disk other than the one you are verifying. Note that there is a space and a comma immediately preceding the DRxx, but no space after the comma.

DO NOT ABORT!
At the completion of Step 9, go on to Step 16.

10. To VERIFY the SYSTEM disk, you MUST direct the VFY scratch file to another disk.

11. If your SYSTEM disk is DRA0: and an available disk where you can write the scratch file is DRZ9:, do the following steps:

```
$ MCR VFY2 ,DRZ9:=DRA0:
```

(Remember the comma before the DRZ9.)
(Read Step 6. for a description of the output generated.)
13. If there are lost blocks do:
   \$ MCR VFY2 ,DRZ9:=DRA0:/LO

14. If there are no multiply-allocated blocks, recover lost blocks by:
   \$ MCR VFY2 ,DRZ9:=DRA0:/RE

15. If there are blocks marked free but actually allocated do:
   \$ MCR VFY2 ,DRZ9:=DRA0:/UPD

   **DO NOT ABORT ANY OF THE ABOVE STEPS!**

16. After finished with VERIFY and if there were any lost files found, you
    should inform the users of the lost files found. There is a script written
    to do this. Get into the shell and type

   
   vfyfini dra0
   if you verified dra0 or
   vfyfini dra1
   if you verified dra1 or
   vfyfini drc0
   if you verified drc0 or
   vfyfini drc1
   if you verified drc1.

17. The final step after finishing with VERIFY is to reboot the VMS system.

   \$ @SHUTDOWN

   When SHUTDOWN finishes, it will type out the following message:

   _SYSTEM SHUTDOWN COMPLETE - USE CONSOLE TO HALT SYSTEM_

   Now you hit "CTRL-P" and the computer responds with:
   >>>
   Now you type in:
   HALT
   The response is:
   >>>
   You type:
   @DBOGEN
   The response is:
   SYSBOOT>
   You type in:
   USE CALSYS.PAR
   The response is:
   SYSBOOT>
   You type in:
   CONTINUE
FILES
None

SEE ALSO
VAX Utilities Reference Manual, Ch. 6.

DIAGNOSTICS
Error messages given are listed in the VAX Utilities Reference Manual, Ch. 6.

AUTHOR
Sheldon Wong

BUGS/DEFICIENCIES
None
NAME
1stdif - compute first differences on a file of data

SYNOPSIS
1stdif [file]

DESCRIPTION
1stdif computes 1st differences (the difference between 2 successive values) on a file of data. If 'n' lines are input, 'n-1' lines are output (there is no difference associated with the 1st value in the file).

Values are free format real numbers. Leading white space is ignored. Blank lines, comment lines (lines that start with a "#") and "missing" values (the value 1e30) are ignored. Only the 1st value on a line is used; remaining characters on the line are ignored.

If no filename is given, standard input is read.

Higher order differences than the 1st can be had by cascading: I.e., the 2nd difference on 'file' would be:

1stdif file | 1stdif

FILES
None.

SEE ALSO
dc(tool), graph(analysis)

AUTHOR
Van Jacobson

BUGS/DEFICIENCIES
NAME
t - Fast Fourier Transform

SYNOPSIS
t [-afit -p pcnt -r rate] [file]

DESCRIPTION

This tool uses the Cooley-Tukey algorithm to calculate the (discrete) fourier transform of the values contained in 'file'. If 'file' is absent, input is taken from standard in.

It is implicitly assumed that the values in 'file' are complex numbers represented by pairs of real numbers, one pair per line. The number of lines in the file must be an integer power of 2. If the file is not already that length, it is "padded" with zeros to make it so. (See the discussion of "padding" under "BUGS" below). In the usual case, the data are reals and if the second element (imaginary part) of a pair is missing, it is assumed to be zero.

The default output is the "power spectrum" of the input data, one value per line, preceded by the corresponding frequency. The mean value (dc component) of the input data is removed first. The power spectra are the squares of the magnitudes of the fourier coefficients. This form is made the default since it seems to be the one most frequently used. If phase information is desired, use '-f' (see below).

Options currently available through flags on the command line are as follows:

- f is used to specify that the output will be the (complex) fourier coefficients, expressed as pairs of reals, one pair per line.

- i specifies that the inverse transform is desired.

Less common flags, used only with the (default) power spectrum output are:

- a to specify that as many values (n, say) will be output as are in the input file. The default is to produce only the first \( n/2 \), since the higher frequencies are aliased.

- t to specify that the mean of the data is to be subtracted from the data before the power spectrum is computed.

- p to specify the amount of tapering. The next argument must be a number between 0 and 100, which is the percentage of the data to be tapered. Tapering, discussed in Chapter 5 of Bloomfield's book (see below) is a technique that smooths the power spectrum by reducing "leakage", a distortion of the spectrum due to dominant frequencies. When in doubt, use the default (which tapers 5% at each end). If the output looks noisy near strong peaks, increase the tapering percentage.
-r must be followed by a positive number, which specifies the sampling rate (# samples/unit time). The transform cannot know the sampling rate unless it is given explicitly. If this flag is not present, the frequencies are \(j/n, j = 0, 1, 2, ..., n-1\), where \(n\) = number of input lines (i.e., the rate is taken to be 1 sample/unit time).

Here are some examples to illustrate a few of the options.

Example 1: Suppose 'file' contains the data [1,2,3,4], one number per line. Then

\[
\text{fft -f file}
\]

will produce

\[
[2.5, -0.5 + 0.5i, -0.5, -0.5 - 0.5i]
\]

while

\[
\text{fft -a file}
\]

will produce

\[
\begin{array}{cc}
0 & 0 \\
0.25 & 0.5 \\
0.5 & 0.25 \\
0.75 & 0.5 \\
\end{array}
\]

Example 2:

\[
\text{fft -f anyfile | fft -i -f}
\]

will reproduce 'anyfile' (to within accumulated numerical errors).

Example 3: To calculate the autocorrelation function of the time series data in 'anyfile', use

\[
\text{fft -a file | field '}$2$'$ | fft -f -i | field '}$1$'
\]

I.e., the autocorrelation sequence is the inverse transform of the power spectrum. (This follows from the general theorem that the transform of a convolution is the product of the transforms).
Large trends (relative to fluctuations about the trend) may distort the power spectrum. For tools to help precondition the input data, see section 6 of the manual.

AUTHOR
Ed Theil

BUGS/DEFICIENCIES
Files must have length equal to a power of 2, and currently have a maximum size of 4096. This value may be changed by modifying the 'defined' value MAXDATSIZ. If the input data is not a power of 2, the file is "padded" with zeros automatically to increase its length to the nearest larger power of 2. Usually, adding zeros at the end of the file will not hurt much, especially if the power spectrum is desired (only a phase change can occur with padding). As a general rule, however, if the number of data items is just slightly more than a power of two, it is better to truncate the data than to permit a long stream of zeros to be added. Ideally, when using the '-f' flag and padding, dummy variables with more plausibility as data than zeros are desirable. (See the first ref. in 'see also').
NAME
   genlin - generate points on a polynomial line

SYNOPSIS
   genlin [-f first] [-i interval] [-l last] [-n npoints] [file]

DESCRIPTION
   Genlin generates x,y pairs on the line of some polynomial. It is generally
   used in conjunction with pfit and graph to make a graph of the polynomial fit
   to some set of data. It reads the coefficients of the polynomial from "file"
   (standard input is read if no filename is given) and generates 'npoints' points
   on that line to standard out. The order of the polynomial is determined from
   the number of coefficients on standard in: 3 numbers will give a line, 4 a qua­
   dratic, 5 a cubic, etc. (the last number is ignored for compatibility with 'pfit'
   which outputs the chi-sq of its fit as the last number). Genlin outputs each
   point as two real values, X then Y, free format and separated by a tab.

   The flags are:

   '-f' (first) should be followed by a real number which sets the starting
   x value. The default is to start at 0.

   '-i' (interval) should be followed by a real number which sets the inter­
   val between x values. The default is 1.

   '-l' (last) should be followed by a real number which sets the last value
   to output. The default is 99.

   (number) is an integer giving the number of points to output. The
   default is 100.

   Up to three of the f, i, l or n flags above can be supplied, in any order. If all
   four are supplied, an error message will result. If fewer than three are sup­
   plied, the default first, last or npnts (depending on which is missing) is used
   as the missing third.

FILES
   None.

SEE ALSO
   dc(tool), field(tool), graph(analysis), pfit(analysis)

DIAGNOSTICS
   None.
AUTHOR
Van Jacobson

BUGS/DEFICIENCIES
NAME

genord - generate ordinal values for lines

SYNOPSIS

genord [-f first] [-i interval] [file]

DESCRIPTION

Genord prefixes numbers to lines read from a file (standard input is read if no filename is given). It is generally used in conjunction with graph(analysis) to generate an x axis for a set of data values. For each line read, the next line number (a free format real number) followed by a tab followed by the original line are written to standard out.

The flags are:

'\-f' (first) should be followed by a real number which sets the starting value. The default is to start at 0.

'\-i' (interval) should be followed by a real number which sets the interval between values. The default is 1.

FILES

None.

SEE ALSO

dc(tool), field(tool), graph(analysis)

DIAGNOSTICS

None.

AUTHOR

Van Jacobson

BUGS/DEFICIENCIES
NAME

graph - graph data

SYNOPSIS

graph [-tek | -grinnel]
    [-nodate]
    [-t top_title] [-b bottom_title]
    [-l left_title] [-r right_title]
    [-noline] [-line]
    [-sym symbol] [-step symbol_step]
    [-ltype line_type]
    [-col color]
    [-y lower yupper] [-x lower xupper]
    [-xlog] [-ylog]
    [-xcal] [-ycal]
    [-xlin] [-ylin]
    [-xnew] [-ynew] [file ...]

DESCRIPTION

Graph is a tool for graphing file(s) of data. It uses the Tektronix AG-II graphing package and provides almost automatic graphing for a wide variety of applications.

The data points are read as real numbers, either one or two values per line. If two values are supplied, the first is used as 'X' and the second as 'Y'. If only one value is supplied, it is used as 'Y' and the index of the line in the file (the 1st line is 0, the 2nd 1, etc.) is used as 'X'. If there are two values, they must be separated by spaces or tabs. The data is free format (e.g., 100, -3.1, 0.51e-4) and leading spaces or tabs are ignored. Blank lines and comments (lines starting with a '#') are ignored. If either X or Y has the value 1e30, ag2 considers it "missing" and will omit the point from the graph.

The typical use of graph would be something like

    graph data

which would graph the file 'data'. Graph picks limits for the x & y axis so that all of data will appear on the graph and supplies a grid, tick marks and tick labels. If there are multiple files of data to appear on the same graph, the command

    graph dat1 dat2

could be used. Note that the limits of the graph are set by the data in 'dat1' so it's possible for data in 'dat2' to be outside the graph limits. If 'dat2' had the same x axis but different units for the y axis, the command

    graph dat1 -ynew dat2
would rescale the y axis (and put up new tick marks and tick labels) before plotting 'dat2'.

If no filenames are given, graph reads standard input (so graph can be used in a pipe). The filename "-" can also be used to refer to standard input.

All of the flags for graph are optional and are mainly used to supply titling, plotting symbols, etc. In general, the flags apply to the next file plotted and persist until changed. E.g., the command

```
graph dat1 -ltype 1 dat2 dat3
```

will change the plot line type for 'dat2' and 'dat3' but not dat1. Flags and filenames can be freely intermixed.

The available flags and their meaning are:

-tek is used to signal the graph is going to a real Tektronix terminal instead of a Grinnel (the default). This flag changes the screen limits to be appropriate for the 1024 x 780 Tektronix instead of the 1024 x 1024 Grinnel.

-grinnel is used to signal that the graph is going to a grinnel (actually, to anything whose screen size is 1024 x 1024) instead of a tektronix.

If you don't say either '-tek' or '-grinnel', 'graph' picks a default appropriate to the system you're using. On our Vax, which has a Tektronix 4014 but no Grinnel, '-tek' is assumed. On the Modcomp, which normally has a Grinnel but no Tektronix, '-grinnel' is assumed. Thus, you don't normally have to say '-tek' or '-grinnel'. If you do supply one of these flags, it has to be the first flag to 'graph'.

-nodate By default, the current date and time are put just above the upper right hand corner of the graph. If this flag appears, the date will be suppressed.

-t is used to supply a top title line. The next argument is used as the title text. The text is displayed above the graph, centered. If multiple '-t' flags are used, each starts a new line below the previous '-t' text.

-b is used to supply bottom title line(s). It is similar to '-t' but the text goes under the graph.

-l is used to supply left side labelling. It is similar to '-t' but the text is displayed vertically and to the left of the graph.

-r is used to supply right side labelling. It is similar to '-t' but the text is displayed vertically and to the right of the graph.
-noline is used to turn off the line through the data points (i.e., gives you a scatter plot of the data). It's equivalent to saying "-ltype -1".

-line is used to turn on the line through the data points. It's equivalent to saying "-ltype 0".

-sym is used to put a plotting symbol on the graph. The next argument is used as the plotting symbol (see table below). The plotting symbol should be either a single character or the character '#' followed by a symbol number. For example "-sym *" would put the character "*" at each plotted data point. (By default, the plotting symbol is placed at every data point. If some other interval is desired (e.g., every 3rd data point), the '-step' flag should be used.)

-step sets the steps between plotting symbols (see '-sym'). It should be followed by an integer giving the interval. "1" puts the symbol at every data point, "2" puts a symbol at every other data point (starting with the first), etc.

-ltype is used to set the line type for the next graph. The next argument must be an integer giving the line type (see table below).

-col is used to set the color for the next plot. The next argument must be an integer specifying the color (0 to 15).

-yl is used to supply 'y' limits for the graph. The next 2 arguments must be real numbers which are used as the lower and upper y limits of the graph. If this flag doesn't appear, y limits are chosen based on the data.

-xl is used to supply 'x' limits for the graph. It is similar to '-yl'.

-xlog is used to get the x axis plotted logarithmically instead of linearly.

-ylog is used to get the y axis plotted logarithmically.

-xcal is used to get the x axis plotted as calendar (date) data. The input must be in the form "YYMMDD" (e.g., 810619 for June 19, 1981).

-ycal is used to get the y axis plotted as calendar (date) data. The input must be in YYMMDD form.

-xlin is used to undo a '-xlog' and get a linear x axis.

-ylin is used to undo a '-ylog' and get a linear y axis.

-xnew causes new limits to be computed for the x axis when the next file is plotted (the default is to use the limits established by the previous plot).
-ynew is like '-xnew' but affects the y axis.

FILES
/dev/tco is used for all graphics output.

SEE ALSO
dc(tool), histds(analysis), mkhist(analysis), pfit(analysis) The Tektronix AG-II Reference Manual

DIAGNOSTICS
None.

AUTHOR
Van Jacobson

BUGS/DEFICIENCIES
There is a maximum of 8000 data points per file to be plotted.

The graph output goes to wherever TCS on a particular machine sends it. See the "tcslib" manual entry for details. On the Vax, output goes to 'TT:' so you need to run graph from the Tektronix. On the Modcomp, output goes to LFN "TCO" so you need to open TCO to an appropriate device before running graph (e.g., "open -lfn tco /dev/gto").

A bug in the real number read routine (ctor) prevents numbers from starting with a decimal point (i.e., ".5" must be input as "0.5" or "5e-1").
### Line Types

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Solid (default)</td>
</tr>
<tr>
<td>1</td>
<td>Dotted</td>
</tr>
<tr>
<td>2</td>
<td>Dot-Dash</td>
</tr>
<tr>
<td>3</td>
<td>Dashed</td>
</tr>
<tr>
<td>4</td>
<td>Long Dash</td>
</tr>
<tr>
<td>9</td>
<td>Alternate Moves and Draws</td>
</tr>
<tr>
<td>-1</td>
<td>No Line</td>
</tr>
<tr>
<td>-2</td>
<td>Vertical Bar to Each Point</td>
</tr>
<tr>
<td>-3</td>
<td>Horizontal Bar to Each Point</td>
</tr>
<tr>
<td>-4</td>
<td>Point Plot</td>
</tr>
</tbody>
</table>

### Plotting Symbols

<table>
<thead>
<tr>
<th>#0</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Circle</td>
</tr>
<tr>
<td>#2</td>
<td>Cross (like 'X')</td>
</tr>
<tr>
<td>#3</td>
<td>Triangle</td>
</tr>
<tr>
<td>#4</td>
<td>Square</td>
</tr>
<tr>
<td>#5</td>
<td>Star</td>
</tr>
<tr>
<td>#6</td>
<td>Diamond</td>
</tr>
<tr>
<td>#7</td>
<td>Vertical Bar</td>
</tr>
<tr>
<td>#8</td>
<td>Cross (like '+')</td>
</tr>
<tr>
<td>#9</td>
<td>Arrow Below Pointing Up</td>
</tr>
<tr>
<td>#10</td>
<td>Arrow Above Pointing Down</td>
</tr>
<tr>
<td>#11</td>
<td>Inverted Triangle</td>
</tr>
</tbody>
</table>

**Char** (any single ascii character) plots that character, centered at the data point.
NAME
histds - graph a histogram

SYNOPSIS
histds [-tek | -grinnel]
   [-nodate]
   [-t top_title] [-b bottom_title]
   [-l left_title] [-r right_title]
   [-col color]
   [-w bar_width]
   [-y l ylower yupper] [-x l xlower xupper]
   [-xlog] [-ylog]
   [-xlin] [-ylin]
   [-xnew] [-ynew] [file ...]

DESCRIPTION
Histds is a tool for graphing file(s) of histogram data. It uses the Tektronix
AG-II graphing package and provides almost automatic histogram displays
for a wide variety of applications.

The data points are read as real numbers, either one or two values per line.
If two values are supplied, the first is used as 'X' and the second as 'Y'. If
only one value is supplied, it is used as 'Y' and the index of the line in the
file (the 1st line is 0, the 2nd 1, etc.) is used as 'X'. If there are two values, they
must be separated by spaces or tabs. The data is free format (e.g., 100, -3.1,
0.51e-4) and leading spaces or tabs are ignored. Blank lines and comments
(lines starting with a '#' are ignored. If either X or Y has the value 1e30, ag2
considers it "missing" and will omit the point from the graph.

Histds considers each X to be a "bin number" and the associated Y to be the
contents of the bin. A vertical bar is drawn from the x-axis to Y. The bar is
centered at X. If the x values are not equally spaced, the resulting graph will
look weird (a real histogram almost always has a constant bin width).

The typical use of histds would be something like

histds data

which would graph the file 'data'. Histds picks limits for the x & y axis so
that all of data will appear on the graph and supplies a grid, tick marks and
tick labels. If there are multiple files of data to appear on the same graph,
the command

histds dat1 dat2

could be used. Note that the limits of the graph are set by the data in 'dat1'
so it's possible for data in 'dat2' to be outside the graph limits. If 'dat2' had
the same x axis but different units for the y axis, the command
histds(analysis)  
March 25, 1985  
histds(analysis)

histds dat1 -ynew dat2

would rescale the y axis (and put up new tick marks and tick labels) before plotting 'dat2'.

When multiple files are being plotted, histds changes the shading of the histogram bar for each file so the user can distinguish the different histograms. The first file is plotted with +45 deg. shading, the 2nd with -45 deg. shading, the 3rd with horizontal shading and the 4th with vertical shading. If there are more than 4 files plotted on the same histogram, the shading types repeat modulo 4. (Actually, after about 2 histograms, the graph gets so visually crowded that you probably won't want to add any more).

If no filenames are given, histds reads standard input (so histds can be used in a pipe). The filename "-" can also be used to refer to standard input.

All of the flags for histds are optional and are mainly used to supply titling. In general, the flags apply to the next file plotted and persist until changed. E.g., the command

histds dat1 -col 6 dat2 dat3

will plot 'dat1' in white, then 'dat2' and 'dat3' in color 6. Flags and filenames can be freely intermixed.

The available flags and their meaning are:

-tek  is used to signal the graph is going to a real Tektronix terminal instead of a Grinnel (the default). This flag changes the screen limits to be appropriate for the 1024 x 780 Tektronix instead of the 1024 x 1024 Grinnel.

-grinnel is used to signal that the graph is going to a grinnel (actually, to anything whose screen size is 1024 x 1024) instead of a tektronix.

If you don't say either '-tek' or '-grinnel', 'histds' picks a default appropriate to the system you're using. On our Vax, which has a Tektronix 4014 but no Grinnel, '-tek' is assumed. On the Modcomp, which normally has a Grinnel but no Tektronix, '-grinnel' is assumed. Thus, you don't normally have to say '-tek' or '-grinnel'. If you do supply one of these flags, it has to be the first flag to 'histds'.

-nodate By default, the current date and time are put just above the upper right hand corner of the graph. If this flag appears, the date will be suppressed.

-t  is used to supply a top title line. The next argument is used as the title text. The text is displayed above the graph, centered. If multiple '-t' flags are used, each starts a new line below the previous '-t' text.
-b is used to supply bottom title line(s). It is similar to `-t' but the text goes under the graph.

-l is used to supply left side labelling. It is similar to `-t' but the text is displayed vertically and to the left of the graph.

-r is used to supply right side labelling. It is similar to `-t' but the text is displayed vertically and to the right of the graph.

-col is used to set the color for the next plot. The next argument must be an integer specifying the color (0 to 15).

-w is used to set the histogram bar width. It should be followed by the histogram bar width in the user’s data units. If `-w' is omitted or negative, the bar width is set to the difference of the 1st two X's of the next data file. I.e., if 2 histograms with different bin widths were to be plotted on the same graph, the command

```
histds dat1 -w -1 dat2
```

could be used to get the right bin width for dat2. In practice, the resulting graph is so "busy" visually that it's better to make separate histogram displays.

-yl is used to supply 'y' limits for the graph. The next 2 arguments must be real numbers which are used as the lower and upper y limits of the graph. If this flag doesn't appear, y limits are chosen based on the data.

-xl is used to supply 'x' limits for the graph. It is similar to `-yl'.

-xlog is used to get the x axis plotted logarithmically instead of linearly.

-ylog is used to get the y axis plotted logarithmically.

-xlin is used to undo a `-xlog' and get a linear x axis.

-ylin is used to undo a `-ylog' and get a linear y axis.

-xnew causes new limits to be computed for the x axis when the next file is plotted (the default is to use the limits established by the previous plot).

-ynew is like `-xnew' but affects the y axis.
FILES
/dev/tco is used for all graphics output.

SEE ALSO
dc(tool), graph(analysis), mkhist(analysis), pfit(analysis) The Tektronix AG-II Reference Manual

DIAGNOSTICS
None.

AUTHOR
Ed Theil, Van Jacobson

BUGS/DEFICIENCIES
There is a maximum of 8000 data points per file to be plotted.

The user should be aware that leading and trailing zero values are deleted from the input data (so that the graph will have the maximum possible resolution). If the resulting X axis is not to your liking, use '-xl' to get it set right (particularly if multiple data files are being plotted on one graph).

A bug in the real number read routine (ctor) prevents numbers from starting with a decimal point (i.e., "0.5" must be input as "0.5" or "5e-1").
NAME
linreg - do a least-squares fit to multivariate data

SYNOPSIS
linreg [-c] [-s] [file]

DESCRIPTION
Linreg does a least-squares fit of a set of multivariate data. The data is read from 'file' (standard input is read if no file is specified) one data point per line. Each data point is an (n+1)-tuple of real numbers, n independent and 1 dependent value. The lines may be in free format and separated by blanks or tabs. Blank lines, comments and 'MISSING' values (x or y = 1e30) are ignored.

The data is fit to the expression

\[ y = a_0 + a_1 x_1 + a_2 x_2 + \ldots + a_n x_n \]

where 'n' may be any number up to 'MAXVARS' (currently set to 10). There must be at least 'n+1' data points for an n'th order fit.

The output of linreg goes to standard out. The output usually consists of one line per input value in the form:

\(<x_1> <x_2> \ldots <x_n> <y_{\text{fit}}> <y_{\text{dat}}> <y_{\text{resid}}>\)

where <x_1>,...<x_n> are the "x" value at the point, <y_{\text{dat}}> is the "y" value at the point and <y_{\text{resid}}> is the difference between <y_{\text{fit}}> and <y_{\text{dat}}>. The output numbers are free format real values separated by tabs.

If the '-c' (coefficients only) flag is given, only the coefficients of the fit are written to standard out as one line in the form:

\(<a_0> <a_1> \ldots <a_n> <\text{chi square}>\)

where the first n+1 parameters are the coefficients of the linear expression and the final parameter is the reduced chi-square of the fit.

If '-s' is present, each parameter appears on a separate line, followed by its associated standard deviation (standard error). This is useful in deciding whether, in fact, the presence of a coefficient in the fit is statistically significant.

SEE ALSO
2dfit(analysis),
pfit(analysis),graph(analysis),genlin(analysis),fft(analysis),field(tool),
regres(analysis), dc(tool)
AUTHOR
Ed Theil

BUGS/DEFICIENCIES
Although the functional form fitted must be linear in the parameters a0, a1, ..., an, it need not be linear in the data values. Thus, if 'file' contained data of the form

\[
x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ y
\]

and one wished to fit the expression

\[
a_0 + a_1x_1 + a_2x_1^2 + a_3x_3^{0.5} + a_4x_4 = y
\]

Then the "script"

```
field '$1 $1'2 $3'0.5 $4 $6' < file | dc | linreg
```

would do the job.

Variations on this trick can be used to fit other functional forms (e.g., \( y = A \times \exp ( B \times x_1 x_2 ) \) can be fit with a natural log transform and dc to perform the multiplication of x1 and x2).

It would be desirable to implement a '-w' flag to provide statistical weighting of the data.
NAME
mkhist - histogram data

SYNOPSIS
mkhist [-n nbins] [-f firstbin] [-w binwidth] [-p] [file]

DESCRIPTION
**mkhist** constructs a histogram of data read from 'file' (standard input is read if no file argument is given). One real number per line is read, free-format with leading blanks or tabs ignored. There is no limit to the number of input values. Blank lines, comments (lines starting with a "#") and MISSING values (values of "1e30") are ignored.

The limits of the histogram are determined from the max and min of the input data and the number of bins in the histogram (which defaults to 50 and can be set to any number $1 \leq nbins \leq 5000$ via the -n flag). If you want to determine either or both of the histogram limits, the optional flags '-f' and '-w' can be used. The '-f' flag sets the first bin of the histogram and the '-w' flag sets the bin width.

The histogram is output to standard out as 'nbins' lines, 2 numbers per line separated by a tab. The first number is the index of the center of the bin and the second is the number of counts in the bin. If the '-p' flag (percentage) is given, the second number will be in percent of the total counts in the histogram rather than in counts.

A message is written to error out giving the number of values binned, the number of underflows (data values < 'first') and the number of overflows (data values > (first - 1 + nbins * width)).

An example of the use of 'mkhist' is the following script to histogram and display the distribution of program sizes for all programs in /usr/src:

```
listree -p /usr/src | ch '%' 'wc -l' >~/tmp
~$/tmp | mkhist -p | histds -t 'Tool Size Distribution'
rm ~$/tmp
```

FILES
None.

SEE ALSO
graph(analysis), histds(analysis), dc(tool), field(tool), fft(analysis)
DIAGNOSTICS
None.

AUTHOR
Van Jacobson

BUGS/DEFICIENCIES
If the histogram limits haven't been specified, the input file must be read twice, once to get the data min and max and once to histogram the data. In this case, the input file can't be a terminal (since it's not possible to rewind a terminal file).
NAME
  npavg - smooth data using a running average

SYNOPSIS
  npavg [-i interval] [file]

DESCRIPTION
  npavg smooths a set of data by doing an n-point running average. It reads
  lines from a file (standard input if no file given) containing the free format
  data values. Only the first value on a line is averaged. The number is added
  to the running average then the averaged result and the remainder of the
  original line are written to standard out.

  The default averaging interval is 3 points. This may be changed to any inter­
  val < 1000 by supplying the "-i" flag followed by an integer giving the interval.

  For example, if you wanted to 5 point average a set of dvm data from the file
  'dvm' and graph the result. The command would be

  npavg -i 5 dvm | graph

FILES
  None.

SEE ALSO
  dc(tool), graph(analysis)

AUTHOR
  Van Jacobson

BUGS/DEFICIENCIES
  The maximum smoothing interval is 1000.
NAME
pfit - do a polynomial fit to data

SYNOPSIS
pfit [-c] [-o order] [-w] [file]

DESCRIPTION
Pfit does a least squares fit of a set of data to a polynomial. The data is read from 'file' (standard input is read if no file is specified) one data point per line. Each data point consists of an x,y pair of real numbers free format and separated by blanks or tabs. Blank lines, comments and 'MISSING' values (x or y = 1e30) are ignored.

The data is fit to the polynomial

\[ y = c_0 + c_1 \cdot x + c_2 \cdot x^2 \ldots + c_n \cdot x^n \]

where 'n' is the order of the polynomial (which defaults to one, e.g., a linear fit, and may be set to any integer <= 20). There must be at least 'n+1' data points for an n'th order fit.

The output of pfit goes to standard out. The output usually consists of one line per input value in the form:

\[ <x> <yfit> <ydat> <yresid> \]

where \( <x> \) is the x data value, \( <yfit> \) is the fitted polynomial value at \( <x> \), \( <ydat> \) is the actual data at \( <x> \) and \( <yresid> \) is the fit residual (\( yfit - ydat \)). The numbers are free format real values separated by tabs. If the '-c' (coefficients only) flag is given, only the coefficients of the fit are written to standard out as one line in the form:

\[ <c0> <c1> \ldots <cn> <chi square> \]

where the first \( n+1 \) parameters are the coefficients of the polynomial and the final parameter is the reduced chi-square of the fit.

All the data points are normally given equal weight in the fit. If the '-w' (weight) flag appears, the data is statistically weighted (i.e., weighted by the magnitude of the y data).

Pfit is normally used in conjunction with other tools. For example, say you had a file of gas pressure vs. flow data on the file 'g' and you want to fit the data to a line and plot the result together with the actual data points:

\[ \text{pfit g | graph -ltype -1 -sym * g} \]

If the fit was higher order than linear but there isn't enough data to draw a smooth curve, one could use the tool 'genlin':

-1-
pfit -c -o 2 g | genlin | graph -lttype -1 -sym * g

and get a smooth quadratic drawn. If one wanted to look for structure in the residuals of the fit, it could be done with the command

pfit -o 2 g | field '$3 $4' | graph

FILES

None.

SEE ALSO

graph(analysis), genlin(analysis), fft(analysis), field(tool), dc(tool)

AUTHOR

Van Jacobson

BUGS/DEFICIENCIES

Fits the the polynomial

\[ y = c_0 + c_1/x + c_2/x^2 + \ldots \]

can be done if dc is used to invert the x values before and after the fit. E.g.,

field -t '1/$1@t$2' | dc | pfit | @@
field -t '1/$1 $2' | dc | graph

assuming, of course, that none of the x values are zero. Variations on this trick can be used to fit other functional forms (e.g., \( y = A \cdot \exp(B \cdot x) \) can be fit with a natural log tranform).
NAME
regres - correlate variables by linear regression

SYNOPSIS
regres [-l lines] [-r] [file]

DESCRIPTION
Given a multicolumn file of numerical data:

\[
\begin{array}{cccc}
  x_{11} & x_{12} & x_{13} & \ldots & x_{1n} & y_1 \\
  x_{21} & x_{22} & x_{23} & \ldots & x_{2n} & y_2 \\
  \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
  x_{m1} & x_{m2} & x_{m3} & \ldots & x_{mn} & y_m \\
\end{array}
\]

\texttt{regres} performs linear regression on all possible combinations of the x's and y's to identify which of the x's are most strongly (linearly) related to the y values. Comments, blank lines and "missing" data (x's or y's >= 1e30) are ignored.

The results are summarized in a table (written to STDOUT) of the form

\[
\begin{array}{cccc}
  \text{nvars} & \text{Cp} & \text{variables} \\
  n1 & \text{Cp(1)} & x_2 & x_3 & x_5 \\
  n2 & \text{Cp(2)} & x_1 & x_2 & \text{etc.} \\
  n3 & \text{Cp(3)} & x_1 & x_4 & x_6 & x_7 \\
  \text{etc.} \\
\end{array}
\]

The table will have 'lines' rows, ordered by the values of Cp in ascending order. 'lines' is defaulted to 10 if the '-l' flag is absent.

Cp is a measure of goodness-of-fit. The expected value of \text{Cp(j)} is approximately j when the fit is good. Roughly speaking, one is interested in small subsets of the independent variables with small values of Cp.

'\text{nvars}' gives the number of independent variables, always including the fitted constant term. Thus, in the above prototype, \text{n1} would equal 4, and the variables entering into the fit are \text{x2}, \text{x3}, \text{x5} and, implicitly, \text{x0}.

Typically, there may be several combinations with approximately equal Cp. The choice is then up to the data analyst as to which subset(s) are appropriate for further consideration. A simple rule, which will not always work is: choose the combination with the smallest number of variables and smallest Cp. However, to complicate matters, it is possible to have the following situation: for, say a subset of two variables, \text{Cp(2)} = 4.25, while for another subset of four variables, \text{Cp(4)} = 4.45. In this case, it is probably better to choose...
the 4 variable subset, since it has less bias in its estimate.

A useful technique is to graph $C_p$ against $p$, which will allow a quick choice of "good" subsets by visual inspection.

If the '-r' flag is present, the table just described is followed by an array of partial correlation coefficients of the data in the form:

correlation coefficients - independent variables

\[
\begin{array}{cccc}
  r_{11} & r_{21} & r_{22} \\
  r_{31} & r_{32} & r_{33} \\
  r_{n1} & r_{n2} & r_{n3} & ... & r_{nn}
\end{array}
\]

correlation coefficients - (xi,y)

\[
\begin{array}{c}
  r(x_1,y), r(x_2,y), ..., r(x_n,y)
\end{array}
\]

Only the lower left diagonal of the correlation matrix is displayed because the coefficients are symmetric.

The correlation matrix is useful because it helps to identify variables that are strongly linearly related and therefore are likely candidates for either 1) treatment as independent variables in a separate analysis or 2) removal from the current analysis.

$r(i,k)$ near either $+$ or $-$ 1 indicates a strong linear relationship, while values near 0 indicate that the variables are statistically independent.

Once the appropriate subset(s) have been chosen, the fitted coefficients may be obtained using 'linreg' or the other fitting tools listed below. It is advisable to choose several candidates for further analysis, since regres by itself cannot provide complete information. Its purpose is as a first step in weeding out irrelevant or otherwise undesirable variables.

SEE ALSO
linreg(analysis), pfit(analysis), 2dfit(analysis), stat(analysis), field(tool)

DIAGNOSTICS
Warnings appear for too little data, too many independent variables and too much data.

Some representative computation times for this tool, on a Modcomp IV, exclusive of data I/O:

<table>
<thead>
<tr>
<th>Number variables</th>
<th>Number data points</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>13</td>
<td>0.1</td>
</tr>
<tr>
<td>5</td>
<td>120</td>
<td>1.0</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>63</td>
<td>3</td>
</tr>
</tbody>
</table>
7 500 17

AUTHOR
Ed Theil
NAME

stat - compute statistics on a file of data

SYNOPSIS

     [-sum] [-count] [file ...]

DESCRIPTION

This tool computes various statistics on a file of data. The file is assumed to
contain free-format real numbers. The first number on each line is added to
the statistics kept for the file. Lines which don't start with a number are
ignored. When the end of file is reached, one line giving the user requested
statistics is printed on standard out. The statistics which can be requested
are:

'avg' The arithmetic mean of the data.

'dev' The population standard deviation of the data. (The square root
     of the variance). The sample standard deviation of the data.

'min' The minimum value of the data.

'max' The maximum value of the data.

'sum' The sum of the data values.

'count' The number of data values accumulated.

The statistics are output in the order that the user requested them,
separated by a tab. E.g., "stat -avg -dev -count" will output one line contain­
ing the average, a tab, the standard deviation, a tab, then the number of
items averaged.

If the argument 'lines' is given, the statistics are accumulated on each line
of the file (i.e., there are multiple numbers on each line, separated by blanks
or tabs) and the requested statistics are printed for each line. Nothing is
printed for lines which contain no numbers.

"Missing" data values (values of 1e30) are not included in the statistics.

If no files are specified on the argument line, standard input is read.

FILES

None.
SEE ALSO
dc(tool), field(tool), lam(tool), graph(analysis), npavg(analysis)

DIAGNOSTICS

AUTHOR
Van Jacobson

BUGS/DEFICIENCIES
NAME
Edit - A Primer Article on the Software Tools Editor

SYNOPSIS
ed [-] file

DESCRIPTION

INTRODUCTION
This document is a detailed description for beginners on how to use the editor called "ed". The editor presented here is modeled closely after a long family of conversational text editors that have achieved wide acceptance. Concern for human engineering dominates the design - ed tries to be concise, regular, and powerful. Because ed is primarily intended for interactive use by programmers, it is streamlined and terse, but relatively easy to use. Ed is not confined to conversational editing, however. It can be driven from prepared scripts and thus is frequently used to select results from programs or to prepare input to still other programs. It is a tool.

GETTING INTO THE EDITOR

To enter the editor, type 'ed' followed by the name of the file to be edited. If a new file is to be started, no name need be specified. Thus one can enter the editor by the following:

% ed file
or
% ed

(The "%" is the shell's prompt.) The first example tells the editor that you wish to edit the file called "file". The second example asks the editor to start from scratch to create a file.

You are now in the editor. A copy of the text is worked on by the editor in a temporary storage area called the "buffer". The editor will attempt to read in the specified file into this buffer. If successful, the editor will inform you of the number of lines read. (A question mark indicates that the file had no lines in it.) You are initially positioned at the END of the buffer. If no file was specified, the buffer is empty. (Note—you are NOT automatically put into input mode if you are creating a file.)

This editor is primarily designed for program development and is thus generally used by programmers, as opposed to inexperienced users. Thus prompting and error messages are kept to a minimum.
COMMANDS

The editor prompts for a command with a colon (":"). All editor commands have the same basic form. A command starts with zero, one, or two line addresses separated by a comma and followed by a command letter. Thus the general format of a command looks like this:

:[line address],[line address]command<param>

The line addresses do not always have to be supplied, but some line address is always assumed. The "command" argument is a one-character letter specifying the command desired. <param> specifies possible additional information needed by the various commands.

Since the editor is primarily designed for interactive use, it is streamlined and terse. Error messages are limited to the simple "?".
LINE ADDRESSING

The editor is oriented around the basic unit called the line. There are three ways to specify the address of a line. The first way is to find the line number of the line you are interested in; the second way is to figure out how far away the line is from the current line; and the third way is to go to the line directly by specifying all or part of its contents.

You are always positioned at some particular line, generally referred to as "the current line". The current line has the name ".", to allow it to be easily referenced. The actual number of the current line can be found by typing ":=" as a command to the editor, or by using the "n" command (see below). Whenever a command is given with no line address preceding it, the current line address is assumed.

Another common and easy way to address a line is by relative addressing. Any line address which begins with a plus "+" or minus sign "-" will be assumed to refer to lines above or below the current line. Thus "+2" would set the current line to be two lines after the current line. Likewise "-5" would set the current line to five lines above the current line. Specifying a line address by itself like this will cause the line at this address to become the current line and to be printed.

You can immediately position yourself at any particular line by simply typing its number. For example, typing

`:1`

would position you at the top of the file. (The colon in this and all subsequent examples is the editor's prompt, and is not typed by the user.) Whenever a line number is given in this manner the line will also automatically be printed.

If you wished to position yourself at the end of the file, and you didn't happen to know the last line number, you could use the symbol ":$", which is used to refer to the last line of the buffer. Thus,

`:$

would cause the last line to become the current line, and the line would be printed.
The last way to specify a line is to search for a particular pattern in that line. This feature will be described in the section on pattern searching.

Line numbers are separated by commas or semicolons; if a semicolon is used, the current line is set to the previous address before the next address in interpreted. This feature can be used to determine the starting line for forward and backward searches.
There are three commands which allow you to add lines of text to the buffer. Each of these commands will place you into what is called the "input mode". Input mode allows you to type in text without its being interpreted as a command. No prompting is done when you are in input mode. To escape from input mode a period "." must be typed as the only character on a line.

**APPEND**

The first of these commands is "a" - append. "A" allows you to add one or more lines of text to the buffer. The general format is:

```
:[line]a
<text>
```

If no line address was supplied, the text is added after the current line. Otherwise, it is added after the line specified. The following is an example of how to use the append command:

```
:a
  For score and seven years ago
    Our fathers give us this great land.
    - John F. Kennedy
```

(The errors in the above example were intentional and will be corrected later.) Remember that to escape from the input mode a single period must be placed on a line by itself.

**INSERT**

The second of these commands is "i" - insert. "I" allows you to put one or more lines of text BEFORE the line address specified, as opposed to after the line as the append command does. The general form of the insert command is:

```
:<line>i
<text>
```

The line address is not required; if it is not supplied it is assumed to be the current line. Thus to insert a heading and two blank lines before the first line
of the above example, we would type:

:1i
    Gettysburg Address

Note that this command also placed you in input mode where you added the heading and the two blank lines and then escaped from input mode by typing a single period on a line by itself. Thus, the buffer now looks like this:

    Gettysburg Address

For score and seven years ago
Our fathers give us this great land.
-John F. Kennedy

CHANGE

The third command is "c" - change. "c" allows you to replace one or more lines with some number of new lines. The general format of the change command is as follows:

    :[line],[line]c
    <lines of text>

The first line address is the first line of text to be replaced, and the second line address is the last line of text to be replaced. To illustrate, the following example shows how to change the first through the tenth line of the buffer to a new section of text:

    :1,10c
    <lines of text>

If only one line address is specified then only that line is replaced. If no line addresses are supplied, the current line is replaced by the text. Thus to change "- John F. Kennedy" to "- Abraham Lincoln", we can do the following:

    :$c
    - Abraham Lincoln

The dollar sign "$" signifies the last line of the file, which was "- John F.
Kennedy". Thus, the buffer now looks like this:

Gettysburg Address

For score and seven years ago
Our fathers give us this great land.
- Abraham Lincoln

Note that after each of these commands the current line becomes the last line added or the last line address specified if nothing was added.
REMOVING TEXT
(d command)

As well as being able to add text, it is just as necessary to be able to delete text. The "d" - delete command allows you to delete one or more lines. The general format of the "d" command is:

:\[line\].\[line\]d

The first line address is the first line to be deleted, and the second address is the last line to be deleted. If only one line address is supplied then only that line is deleted. If no line addresses are supplied, the current line is deleted and the line immediately FOLLOWING the deleted section becomes the new current line. Thus, deleting one of the blank lines underneath the header can be accomplished by:

:\-3dp
For score and seven years ago

Thus the buffer now looks like this:

Gettysburg Address

For score and seven years ago
Our fathers give us this great land.
- Abraham Lincoln

For illustration, let us say that if we had decided to delete the header entirely, we could have done this as follows:

:\1,3dp
For score and seven years ago
MOVING TEXT
(m and k commands)

Lines of text can be moved from one place to another with the "m" - move - command. The general format of the "m" command is:

:[line],[line]m[line]

The first and second line addresses are used in the same manner as in the previous commands. The third line address specifies which line to place the moved text after. Thus, moving "- Abraham Lincoln" to before the quote can be accomplished by:

:$m2p
- Abraham Lincoln

This command moved the last line of the file so that it now follows the second line of the file. The "p" caused the new current line to be printed. Note again that the dollar sign "$" represents the last line of the file. After the command, the last line moved becomes the current line. Thus, the buffer now looks like this:

Gettysburg Address

- Abraham Lincoln
For score and seven years ago
Our fathers give us this great land.

"k" - copy - command. The "k" command is similar to the "m" command, except that it copies a block of lines from one location to another, thereby duplicating text. The general format of the "k" command is:

:[line],[line]k[line]

where each line address is used just as they are with the "m" command. The heading of our quote could be duplicated by:

:1,3k3

This command copied the first three lines to the third line in the file. The buffer now looks like:

Gettysburg Address
Gettysburg Address

- Abraham Lincoln

For score and seven years ago
Our fathers give us this great land.

PRINTING TEXT
(p, l, n, b, =, and <CR> commands)

The command "p" - print - is used to cause lines to be printed. The general format is:

:\[line],[line]p

If no line addresses are given the current line is printed. The "p" command can also be appended to most other commands to cause the last line affected by the command (the new current line) to be printed. For example,

:\1,2m$\p$

would move lines 1 and 2 to the end of the file and print the last line moved (now the last line of the file).

The "l" - list - command has the same format as the "p" command ([line],[line]l), but instead of simply printing the specified lines, it lists them unambiguously. This means that non-printing characters such as control characters are printed with a preceding caret ("~") and then a corresponding printing character. For example, on most machines the tab character is <CTRL>I. The "l" command would print this as:

\~l

The "l" command also indicates where the last character of a line is by printing a dollarsign ("$") after the final character. Thus a line like:

hello<CTRL I>dolly<SP><SP><ENF OF LINE>

might be printed with the "p" command as:

hello   dolly

but the "l" command would list it as:

hello~Idolly $
The "n" - number - command has the same format as the "p" command ([line],[line])n, and is the same as the "p" command in function except that it preceeds each line with the line's line number. Back to our Gettysburg Address example, the command:

:1,$n

would result in the output:

1
2 Gettysburg Address
3
4
5 Gettysburg Address
6
7 - Abraham Lincoln
8 For score and seven years ago
9 our fathers give us this great land.

The "b" - browse - command has the format:

:[line]b[+-]

The "b" command prints a screenful of text (or the entire file, whichever is less), using the specified address, if any. If no address is specified, the "b" command uses the current line. The "b" command has three forms:

[line]b+
prints a screenful of text starting at [line].
This is the default. The last line printed ([line] + screensize) becomes the current line.

[line]b-
prints a screenful of text ending at [line].
This means that "b" will back up enough lines so that a screenful of text will end with [line]. [line] becomes the current line.

[line]b. 
prints a screenful of text with [line] in the middle of the screen. Thus, "b" backs up half a screenful of lines and from there prints a screenful. The last line printed ([line] + (screensize / 2)) becomes the current line.
Typing only a carriage return on a line causes the next line in the file to become the current line and to be printed. This is an easy way to step through your file. When you reach the bottom, where there is no "next line", the editor will respond with a question mark "?".

The "=" command prints the number of the current line. For example,

```
:=
5
```

The editor typed the "5" is response to the "=".
PATTERN SEARCHING
(/ and \ commands)

It is possible to search for a certain pattern of characters to find where it occurs in the file. The editor will find the next line containing this pattern and print it. The forward search command is of the following form:

/:<pattern>/

The trailing slash is optional. If necessary the search will wrap around to the top of the buffer. The search can be repeated by:

/:// or just
/:\

These were forward searches; you can also search backwards by using the backslashes "\"

/:<pattern> would search from the current line backwards looking for the pattern. Again, the search will wrap around to the bottom of the buffer if necessary.

The repetition of the backwards search is also similar:

/:\ or just
/: It is possible to indicate that the pattern desired is specifically at the beginning or end of a line. The percent "%" is used to indicate the beginning of a line, and the dollar sign "$" to indicate the end. Thus,

/:%hi/

would find the first occurrence of "hi" at the beginning of a line. Similarly, the following would look for the first occurrence of a line ending with a "z":

/:z$/

If a search fails to find the pattern specified, the editor will type a question mark "?".
PATTERN SUBSTITUTION
(s command)

The "s" - substitute - command allows you to replace one or more characters on a line with zero or more other characters. The general format of the "s" command is:

```
:line1,[line2]s[/<pattern>/]<string>
```

where "pattern" stands for the characters to replace, and "string" the characters to replace them with. The first and second line addresses are used as in previous commands. The delimiter slashes "\" can be replaced with any character which is not part of the pattern or the following string. The <pattern> is the same as used in the searching command but in this case it is searching the line, rather than the file, for this pattern. Only the first occurrence of the pattern will be substituted (unless the "g" command is used—see below).

Following are some examples of pattern search in conjunction with substitution:

```
:/ForlsjFor/Fourl
```

or

```
:/For/s//Four/
```

In the first example, we first search for the line containing the pattern we are looking for ("/For/"). Then we replace in that line the word "For" by "Four". The second example is more interesting and illustrates a time-saving feature of the editor. In the second example, we search for "For" as we did in the first example, but now instead of explicitly saying what the pattern is, we leave it empty. When the editor finds the search pattern empty it will assume it to be the last pattern searched for using the pattern searching facilities. Thus, in this case the editor would remember that the pattern to be replaced is "For".

A substitution pattern can also be repeated verbatim by using the character "&". This character stands for "whatever was found and matched". Thus,

```
:/For/s//&e/p
```

Fore score and seven years ago

would locate "For" and append an e to it.

If no /string/ is given, the /pattern/ matched is deleted. On our current line,
:s/score /
Fore and seven years ago

The word could be put back by:

:s/ /score /
Fore score and seven years ago

It is sometimes necessary to insert or delete newline (the end-of-line indicator) characters in lines, thus breaking or concatenating them. This can be done by using the convention "@n" to stand for the newline character. For example, if you wished to change the line "Four score and seven years ago" into two lines, you could do this:

/:Four/s/score/score@n/
:^1..p
Four score
and seven years ago

To then put the lines back together again, try this:

/:Four/s/@n//

If you type the "s" command with neither a /pattern/ or a /string/, the effect is to substitute the last /string/ for the last /pattern/ (whether it was mentioned in an "s" command or in an expression search). For example:

:s/e/x
Four scorx and seven years ago
:s
For scorx and sxven years ago
:/us/s
our fathers give x this great land.

The "s" command may be followed by the "p" (print) or followed or preceded by the "g" (global--see below) commands whenever desired.
Note that inter-line and intra-line searching stops at the first occurrence. It might be nice to find all occurrences in the line or find every line which has a pattern in it. Thus the "g" - global - command comes in handy.

The "g" command can be used in two difference ways: "globally throughout the line" or "all lines which have this". The "g" command is most often used along with a substitute command but this is not the only command it can be used with. If you wished to search for all lines containing a certain pattern, the following is the general format:

:g/<pattern>/command

To make the "s" command globally find and replace the specified pattern throughout the line, the general form is:

:[line],[line]s/<pattern>/<string>/g

You can also combine these two forms to change every occurrence in the file. An example of these forms combined is:

:g/give/s//gave/g

The above example would search all lines for the pattern "give" and change every occurrence to "gave".

You can also execute more than one command for each global match. To do this, add an atsign ("@") to the end of each command except the last. For example,

:g/pattern/s//otherstuff/@
 s/%/###/

This would change the first occurrence of 'pattern' in every line to 'otherstuff' and flag the beginning of the line with '###'.


It is also possible to search for lines which contain any patterns other than those specifically given. This can be done by using the "x" - except - command. The format is:

:x/<pattern>/command

The x command is the same as the g command, except that lines NOT containing the pattern are located.

For example, suppose the file being edited contained a list of filenames from your library. Perhaps you might only be interested in files which contained a certain string of characters, like "test" (such as "retest", "newtest", "iotest", etc.) Then you could delete all other file names by using the "x" command:

:x/test/d

The above command would search through the entire file and delete any line which did NOT contain the character string "test".
DEALING WITH FILES
(w, r, e, and f commands)

W (WRITE) COMMAND

Now that you know how to create and change your masterpiece, you would probably like to save it. Since it saves the contents of the buffer for posterity, the "w" - write - command is an extremely important command. The general format is:

:\[line\],[:line]\w <file>
or

:\[line\],[:line]\w

If no line numbers are specified, the entire buffer is written. When the "w" command is invoked without specifying the file's name, the name last associated with the buffer is used (such as the name you gave when you entered the editor). If no name has been associated with the buffer, a file name must be supplied. Thus to save our example for later use as "test", the following form of the "w" command would be used:

\[ Gettysburg Address \\
5 - Abraham Lincoln \\
6 For score and seven years ago \\
7 our fathers give us this great land.

:w test  
"test" 5 lines, 116 characters

After this command has been given, the name specified is now associated with the buffer and is usually called the "remembered" file name. If you do some changes to this file and then wish to write out this new version, this can be accomplished by simply using the "w" command with no name specified:

:w  
"test" 5 lines, 116 characters

It is also possible to write out a portion of the buffer to a file. To write out the last two lines into a file called "new", you would do the following:

:4,5w new 
"new" 2 lines, 67 characters

R (READ) COMMAND

April 11, 1985
It is also possible to read a file into the buffer at the location you specify. This is done by using the "r" - read - command. The general format is:

[line]r file

If no line number is given, the file specified by "file" is read into the buffer after the current line. If a line number is given, the file is read into the buffer after that line.

The following is an example of how the "r" command is used:

:r x
"x" 4 lines, 96 characters

The above command reads in the file "x" and inserts it after the current line. If you had wished to read this file in at another line address, say after line 3, you could have done it this way:

:3r x
"x" 4 lines, 96 characters

The read command also allows line zero ('0') to be specified, indicating that the file is to be read in at the very beginning of the file, before any other text.

If no file name is given, the "remembered" file name is used.

E (EDIT) COMMAND

The "e" - edit - command deletes the contents of the buffer and reads a new file into the buffer. The "e" command does NOT check first to see if you have saved your current file. The general format is:

:e <file>

If no file name is specified then the current file is read in again. For example,

:e x
"x" 4 lines, 96 characters

The above command deletes the current buffer, reads in the file named "x", and prints the number of lines found. Giving this command is the same as entering the editor as you did initially.
F (FILE) COMMAND

The "f" command is useful for printing the name of the current file (in case you've forgotten it) or for changing the remembered file name. The general format is:

    :f
or
    :f <file>

If no file name is given, the editor will print the name of the current file. If a file name has been specified, the name associated with the buffer will be changed to whatever you typed. Thus to change the file name from "test" to "ambrose", do the following:

    :f ambrose

This command does not alter the state of the buffer nor does it write the buffer out. It also does not affect the name of any actual file on disk—it simply resets the name of the file you plan to write the buffer to.
EXITING THE EDITOR
(q, q!, and wq commands)

The "q" - quit - command is used to leave the editor. The format is:

:q

The editor will check to see if you have saved your file, and if you haven't, will warn you by printing:

No write since last change (:q! overrides)

The "q!" - quit! - command tells the editor that you want to quit even though you haven't saved your file. The format is:

:q!

The "wq" - writequit - command has the format:

:wq [filename]

Its effect is to do exactly equivalent to:

:w [filename]
:q
REGULAR EXPRESSIONS

Searching for literal strings (with the "s" command, for example) is convenient, but you will soon find it restrictive. Accordingly, the editor 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-searching capabilities include a class of patterns called "regular expressions". Regular expressions typically include a way to specify alternates and the ability to parenthesize patterns. To be able to express these more general patterns, some special characters (often called metacharacters) are used.

MATCH ANY CHARACTER

The character "?" matches any single character (except a newline). Thus, if you wanted to search for a string that contained x, anything else, and y you could do it this way:

```/x?y/```

The pattern "x?y" matches x+y, xay, x y, and similar strings.

CHARACTER CLASSES

The character "[" signals that the characters following, up to the next "]", form a character class, that is, a text pattern that matches a single character from the bracketed list. Thus, [aA] matches "a" or "A", [a-z] matches any lower case letter, and [eqp] matches either e, q, or p. So, to search for a pattern which starts with x, ends with y, and contains either +, z, or a space between them, you could do:

```/x[+z ]y/```

Or, if you wished to search for a pattern which starts with x, ends with y, and contains any letter (but not number or special character) between them, you could do this:

```/x[a-zA-Z]y/```

NEGATED CHARACTER CLASSES

It is also possible to search for lines which contain anything but a specific set of characters. The exclamation point "!" indicates that searching should be done for any characters other than those in the class. For example:
would locate any line which contained any character other than a number. And,

`:/[!a-zA-Z]/`

would find any line which contained anything besides the capital or small letters.

Note that this command does not exclude lines if they contain the characters you specified; that is, if you gave the command

`:/[!0-9]/`

and the next line in the file was "a9", that command would match that line since it contained a character ('a') other than 0-9. To actually exclude line matches, using the "x" command.

**Closures**

Any of the text patterns above that match a single character can be followed by the character "*" to make a text pattern that matches zero or more successive occurrences of the single character pattern. The resulting pattern is called a closure. For example, a* matches zero or more a's; aa* matches one or more a's; [a-z]* matches any string of zero or more lower case letters. Also, [a-zA-Z]* matches an entire word (which may be a null string) and ?* matches a whole line (which also may be a null string). And you might like to use (?)* to match anything between parentheses, or [a-z][a-z0-9]* to match the leftmost Fortran identifier on a line. Closures are a very powerful tool.

**Beginning and End of Lines**

Two other metacharacters do not match literal characters but rather match positions on the input line. "%" matches the beginning of a line: %abc is a pattern that matches abc only if it occurs as the first three characters of an input line. Analogously, $ matches the newline (line feed) at the end of a line: abc$ matches abc only if it is the last thing on a line, before the newline. Of course these can work together: %abc$ matches line that contains only abc and %$ matches only empty lines.

**Ditto Character**

The ampersand "&" can be used to repeat what was matched. That is, the ampersand means "whatever was found". For example,
`:s/book/&s/`

would append an "s" onto the end of the pattern "book".

**ESCAPE CONVENTIONS**

Sometimes it is necessary to search for characters which are the same as metacharacters. Thus the special meaning of any metacharacter can be turned off by preceding it with the escape character "@". Thus @? matches a literal question mark and @@ matches a literal atsign.

**SUBPATTERNS**

A pattern can be broken into several subpatterns that can be rearranged into a different pattern by using the "@(" and ")@" features. Each section of a pattern that is enclosed by "@(" and ")@" can be referred to in /string/ by @n, where 'n' is the number of the subpattern, starting with zero (0). For example, on the line:

```
Pattern rearrangement example
```

you can type:

```
:s/at@(a-z)*@/x@Ox
```

```
Pattern rearrangement example
```

To reverse the second and third words:

```
:s/@(rearrangement@)/@1@0/
```

```
Pattern rearrangement example
```

The important thing to remember is that the first subpattern is referred to as "@0" and NOT "@1".

**REPEATING COMMANDS**

The "&" command has the format:

```
:&
```

It simply repeats the last command, *verbatim*. No line numbers can be given to this command. On a file such as:

```
This is line 1 of a file to show how & works
Being a line in an example file is a drag
The final line
```

```
-24-
```
The following dialogue could occur:

```
:1
This is line 1 of a file to show how & works
:s/is/
This line 1 of a file to show how & works
:&
This line 1 of a file to show how & works
:w tmpfile
- file tmpfile: 3 lines
:&
- file tmpfile: 3 lines
:&
- file tmpfile: 3 lines
:& tmpfile
?
...+1n
 1 This line 1 of a file to show how & works
 2 Being a line in an example file is a drag
:&
 2 Being a line in an example file is a drag
 3 The final line
:&
?
```

The first "?" (in response to the command "& tmp2file") occurred because the "&" command repeats the last command exactly, and therefore does not allow any additional parameters. The final "?" occurred because ed tried to do ".,+1n" starting at line 3, which is equivalent to "3,4n". There is no line 4 in the file, hence the error. If there were, the second "&" command would have been perfectly legal.

COMMENTS

Any command that starts with a pound sign ("#") is ignored by ed. This feature allows easy commenting of scripts (see READING FROM SCRIPT FILES below).

SPAWNING TO THE SHELL

The "!" command has the format:

```
:! [<shell command>]
```

Its action is to take the line following the "@" and to spawn a shell with that line as the shell's arguments. For example, the following command would format and print this document:

```
:!m /usr/man/tutorial/edit.m | lpr
```
Note that, unlike the shell, ed does NOT recognize a final "@" at the end of a line as specifying line continuation. The following command

:!cat jim ; @

will spawn a shell with "cat jim ; @" for a command.

If the current file has not been written out since its last change, ed will warn you by typing:

[No write since last change]

This statement is a reminder that the latest version of your file exists only in memory, and that any reference to that file by your shell command will process the previous version of your file (if any), and not the latest, corrected version.

A lone "!" on a line will spawn an interactive shell. To exit the shell and return to the editor, type:

% logout

READING FROM SCRIPT FILES

It is sometimes convenient to read edit commands from a file rather than from a terminal. This capability is frequently used to select results from programs or to prepare input to still other programs. To have the editor read its commands from a file, do the following:

% ed - <infile file

The "<infile" means substitute the file named "infile" for the standard input file, which is generally the teletype. The file "infile" should contain whatever edit commands you need for accomplishing the task you had in mind.

You might like to combine processing from the teletype and script files, such as having a user fill in the spaces on a simple form. For example, if you prepared a script file such as this:

a
PROBLEM REPORT
General form for reporting problems
Name: 
Date: 
Nature of Problem: 

3p
r /dev/termin
+1p
r /dev/termin
+1p
r /dev/termin
w report
q

Where /dev/termin is the name of the user's input device. The user could then invoke the editor and receive prompts for information desired in the form. For example, if the above file were called "script" then the session would look like this: (prompts given by the editor are preceded by ">" for clarity)

:ed - <script
> Name:
  Gwendolyn Arnold
  (end-of-file character)
> Date:
  April 24, 1978
  (end-of-file character)
> Nature of Problem:
  I would like to report a malfunction...
  <rest of problem>
  (end-of-file character)

The "end-of-file character" indicates that the user must indicate an end to her input by sending the end-of-file character. The above completed script form would be written to file "report", which the user could then send to whoever might be interested.
EXAMPLE OF A SIMPLE EDITTING SESSION

The following is an example of an editing session:

```
% ed
:a
For score and seven years ago
Our fathers give us this great land.
- John F. Kennedy

:i

Gettysburg Address

:c
- Abraham Lincoln

:3dp
For score and seven years ago
:sm2p
- Abraham Lincoln

:1, $p

Gettysburg Address

- Abraham Lincoln
For score and seven years ago
Our fathers give us this great land.

:/For/
For score and seven years ago
:give Our fathers give us this great land.
:/For/s/ Four /p
Four score and seven years ago
:g/give/s/ gave/g

:1,$p

Gettysburg Address

- Abraham Lincoln
Four score and seven years ago
Our fathers gave us this great land.
:w test
"test" 5 lines, 118 characters
:# I wonder if I've put in eight hours yet
:!date
07Jul91 15:22:05.37
:# Nope
```
Gettysburg Address

- Abraham Lincoln
For score and seven years ago
Our fathers gave us this great land.
COMMAND SYNOPSIS

Here is a summary of the commands and facilities of ed.

Commands may be preceded by none, one or two line numbers. The one or two are used as needed. If line numbers are not specified, defaults are used whenever possible. Line numbers are formed from the following components:

- 17: an integer number
- .: the current line
- $: the last line
- +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 are separated by commas or semicolons; a semicolon sets the current line to the most recent line number before proceeding.

The following general commands are recognized by the editor:

<table>
<thead>
<tr>
<th>Default</th>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td>Lines</td>
<td>Lines</td>
</tr>
<tr>
<td>(.)</td>
<td>a</td>
<td>Append text after line (text follows)</td>
</tr>
<tr>
<td>(.)</td>
<td>b[-+]</td>
<td>Browse through a screenful of text</td>
</tr>
<tr>
<td>(...)</td>
<td>c</td>
<td>Change text by deleting lines specified and inserting new text in their place (text follows)</td>
</tr>
<tr>
<td>(...)</td>
<td>d</td>
<td>Delete lines specified</td>
</tr>
<tr>
<td>e</td>
<td>&lt;file&gt;</td>
<td>Delete current buffer; read in &quot;file&quot; for editing</td>
</tr>
<tr>
<td>f</td>
<td>&lt;file&gt;</td>
<td>Print file name or change it to &quot;file&quot;</td>
</tr>
</tbody>
</table>
(1,$)  g/<pat>/command  Causes repeated execution of a command, once for each line that matches the specified <pat>. Multiple commands may be executed by terminating all but the last with an atsign ('@')

.  i  Insert text before line specified (text follows)

(.,.)  k line  Copy lines specified after line

(.,.)  l  List lines unambiguously

(.,.)  m line  Move lines specified to after line

(.,.)  n  Number lines as they are listed

(.,.)  p  Print current line, last line affected by command, or lines specified by command

q  Leave editor
q!  Leave editor even if current file has not been saved

(.)  r file  Read file, appending after line

(.,.)  s/pat/new/g  Substitute 'new' for first occurrence of 'pat ('g' implies repeatedly across line)

(.,.)  w <file>  Write buffer to "file"

(.,.)  wq <file>  Write buffer to "file" and quit

(1,$)  x/<pat>/command  Causes repeated execution of a command, once for each line that does NOT match the specified <pat>

! [command line]  Spawn to the shell to execute command line or spawn to interactive shell if no command is given

(,)  =  Print line number

&  Repeat last command verbatim
# Comment

"n"  Print text at line "n"

<CR>  Print next line, which then becomes current line

/<pat>/  Search forward for pattern

<pat>  Search backwards for pattern

.  Leave input mode

The following are special characters used by the editor:

?  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)

*  Closure (zero or more occurrences of previous pattern)

@  Escaped character (e.g. @%, @[, @*)

&  Ditto, i.e. whatever was matched

c1-c2  Range of characters between c1 and c2

@n  Specifies the newline character at the end of a line

@t  Specifies a tab character

@() and @(  Specifies subexpressions
AUTHOR

Major portions of this writeup were adapted from David A. Mosher's "Edit is for Beginners", a tutorial available from the UC Berkeley computer center library. Descriptions of RTSG enhancements to ed were added by Vern Paxson.
NAME

lextut - how to use lex

SYNOPSIS

lextut [-distv] [file]

-d: scanner should operate in debug mode
-i: make a case insensitive scanner
-s: Suppress default rule
-t: Trace processing
-v: Verbose

DESCRIPTION

1. Introduction.

lex generates a program from its input file to perform lexical analysis of text. Such a program is often called a scanner. The input file to lex (standard input is assumed if a filename isn't given) contains regular expressions for which the generated scanner will search text. The input file also contains actions (ratfor statements) which will be executed when their corresponding regular expression is matched.

The output from lex is ratfor code defining the integer function yylex. When the generated program containing yylex is compiled, linked, and executed, it will search its input text for strings matching the regular expressions, and execute the corresponding action. Text that is not matched by any expression is simply copied to the output.

Here is a diagram of the situation:

```
+-------+
| regular expressions, --->| lextut | ---> yylex |
+-------+
| actions | .......................... |
+-------+
| text ----> yylex | ----> output |
+-------+
```

lex is best suited for writing programs which detect non-contextual patterns—that is, patterns whose recognition does not dependant upon what has been seen previously. If you want to be able to detect complicated patterns in context (i.e., based on what's been seen in previously), you probably should use yacc(tool) to recognized the complicated patterns and lex to recognize the simpler tokens from which the more complicated patterns are constructed.
2. Simple rules.

As a trivial example, here is a lex input file which will generate a scanner to remove blanks and tabs at the ends of lines:

```
%%
[@t]+$   # (no action)
```

The line containing only "%%" marks the beginning of the rules. The following line is the program's one rule. The rule contains a regular expression which matches one or more instances of the characters blank or tab (written @t) just prior to the end of a line. The brackets indicate the character class made of blank and tab, the + indicates "one or more", and the $ indicates "end of line". No action is specified, so the routine generated by lex, called yylex, will ignore the matched characters. All other characters will be copied from input to output--that is the default action of lex.

To change any remaining string of blanks or tabs to a single blank, add another rule:

```
%%
[@t]+$   # (no action)
[@t]+   call putc(' ')
```

The scanner generated by this input file will scan for both rules at once, observing at the termination of the string of blanks or tabs whether or not there is a NEWLINE character, and then executing the desired action. The first rule matches all strings of blanks or tabs at the end of lines, and the second rule matches all remaining strings of blanks or tabs.

As a slightly more useful example, suppose you want to change a number of words from British to American spelling. lex rules such as:

```
colour   call putlin( "color", STDOUT )
mechanise call putlin( "mechanize", STDOUT )
petrol   call putlin( "gas", STDOUT )
```

would be a start. These rules are not quite enough, since the word petroleum would become gaseum. To handle cases like that correctly, you would need more complicated expressions.

3. More complicated expressions.

Regular expressions in lex are similar to the regular expressions in ch, find, and ed. There is a common set of operators which are exactly the same in all four programs. These are documented in the tutorial on regular expressions (see regexp(tutorial)). In addition, lex has a number of operators which the other programs don't know about. The operators which are unique to lex are:

```
""   ( )   +   |   {}   /   <>
```
The quotation mark operator, "", is used to quote whole strings just like the @ operator will quote a single character. Whatever is contained between a pair of quotes is just plain text. Thus

```
xyz"[12]"
```
matches the string xyz[12], and not xyz1 or xyz2. Since the []'s are inside quotes, they are interpreted as simple characters instead of character class delimiters. Another use of quotes is to get a blank into an expression; normally a blank would indicate the end of the rule, but if it is inside quotes, it is just another character.

Parentheses are used for grouping, just like in math—they indicate the order of evaluation. They are very useful in conjunction with the other operators. For example, without parentheses, the closure operator * can only apply to single characters and character classes. With parentheses, you can apply it to an entire string, like this:

```
(foo*)
```

This expression will match 0 or more instances of the string "foo".

The plus operator, +, is called "positive closure". It means "1 or more of the previous", just like * means "0 or more". We saw an example of + at the beginning of this document, in the expression to match 1 or more blanks or tabs at the end of lines.

The alternation operator, |, is used to indicate that either one of the alternatives can be matched. For example, to match any one of the strings "foo", "bar", or "bletch", you could say:

```
foo|bar|bletch
```

Parentheses are not needed around the words because alternation has a lower precedence than concatenation. Precedence is discussed in greater detail below.

The optional operator, ,, matches an optionally present expression. For example,

```
[+-][0-9]+
```

matches a string of digits preceded by an optional plus or minus sign. If the sign is present, it is included in the matched text, but if it is not present, the pattern will still successfully match a string of digits.

The curly braces, {}, have two different meanings depending on whether the string they enclose is a name or numbers. Curly braces with a name inside indicate that the name is a definition which should be expanded. For example,
{DIGIT}

would look for a predefined string named DIGIT and insert it at that point in the expression. The definitions are given in section 1 of the lex input, which is described below.

Curly braces with numbers inside indicate a certain number of iterations of the previous pattern—it is a generalization of the * and + operators. There are three different forms:

\[ a[2,5] \]

would match 2 to 5 occurrences of \( a \),

\[ a[2,] \]

would match 2 or more occurrences, and

\[ a[2] \]

would match exactly 2 occurrences.

The / operator is used to indicate trailing context. The expression

\[ ab/cd \]

will match the string "ab", but only if it is followed by the string "cd". Note that the trailing context part—"cd" in this example—is NOT part of the matched string. "ab" is the matched string, and "cd" is pushed back onto the input stream to be matched later.

The $ operator, which matches the ends of lines, is a special case of the / operator. Thus

\[ ab$ \]

is the same as

\[ ab/@n \]

Left context is handled in lex by the <> operator, called a start condition. A start condition is declared in the definitions section. Then, if a rule is only to be matched when the lex interpreter is in start condition \( x \), the rule should be prefixed by

\[ <x> \]

Since start conditions are a little hard to understand, they are covered in more detail in the section on left context.

The lex operators with the lowest precedence are the alternation (|) and trailing context (/) operators. Concatenation is just above these two
operators in precedence. Concatenation is the implicit operator that makes "xy" mean "match an 'x' and then match a 'y'". The unary operators, '^', '+', '-', and '{'} have equal precedence, higher than that of concatenation, and associate from left to right. Operators such as '?' and '[...]'] are identical to simple characters from a syntactic point of view, and as such have no precedence associated with them. Note that meta-characters (except '-' and ']') lose their special meanings inside '[...]]'s and need not be escaped. Parentheses and quoted strings have the highest precedence. Here are some examples of how precedence works:

```
ab[3]|cd*
```

matches either an "a" followed by three "b"s, or a "c" followed by zero or more "d"s.

```
a(bc+)
```

matches an "a" followed by one or more occurrences of the string "bc".

```
a{4}{6,}
```

will match 6 or more groups of 4 "a"s (24 "a"s, 28 "a"s, 32 "a"s, ...).


So far, all we have seen as lex actions are calls to `putc()` and the null action. Actually, a lex action can be any group of ratfor statements. If the statements span more than one line, they must be enclosed by "%~" and "%~".

One thing that is useful for an action to do is to pass a value back to the main program. This can be done by returning it as the function value. For instance, if the lexical scanner generated by lex is being used as the front end of a compiler, then for each token recognized the action might be "return ( TOKENID )", where "TOKENID" identifies the particular token recognized. A value can also be returned by assigning it to yylex's argument, which is an integer variable yyval. In this manner, for example, a scanner for a compiler can return the token type using the "return" statement and the token value in the variable yyval. This choice of interface makes it easy to use lex to generate a scanner for a parser generated by yacc(tool).

If more than two values need to be passed back, the others can be placed in common variables declared in the local definitions section of the lex input. There is a section further on describing this.

There are a number of special routines available to actions. ECHO is a macro which simply writes the current text to STDOUT. It is the default action for characters not otherwise matched. You also might want to call it yourself. For instance, if you wanted to copy a file but also double all occurrences of
the word "rabbit", you might use the rule:

```
rabbit {%ECHO
       ECHO
       ECHO%
```

Rabbit will be echoed twice, and everything else will be echoed once by default.

**YYGETTEXT** is a macro which gets the matched text and stores it into a buffer you supply. This is probably the most useful of the special actions. As an example of its use, here is a lex scanner which turns upper case alphabetic strings into all lower case:

```
%%
character buf(MAXLINE)

[A-Z]+

YYGETTEXT(buf)
    call fold(buf)
    call putlin(buf,STDOUT)

```

**YYFIRSTCHAR** is a macro which returns the first character of the text matched. A sample all is:

```
[a-z]  yyval = YYFIRSTCHAR; return(TOK_LETTER)
```

**YYINPUT** is called by the scanner to get more input. It takes three arguments. The first is an array in which to put the input, the second is the position to start putting the input in the array, and the third is assigned the number of characters that were read, or EOF if an end-of-file was read. This is the most commonly redefined macro. Typically, if you want the scanner to read from some global file descriptor *fd* instead of the standard input, redefine **YYINPUT** (in the global definitions section—see below) like so:

```
define(YYINPUT,$3 = getlin($1($2), fd))
```

Note that the scanner automatically declares the function *getlin*. If you wish to use some other function, you must declare it in the local declarations part of the second section of the lex input file (see below). Note that the scanner returns EOF upon reading an end-of-file, so the routine calling the scanner can detect that the EOF was found and arrange for *fd* to refer to the next file to be processed.

**YYUNPUT** puts its character argument onto the scanner's input stack. This character will be the next one processed, even if the last rule matched
involved trailing context. A sample use is:

```
[0-9] %
  # add leading 'leon' so next pattern will match it
  YYUNPUT(YYFIRSTCHAR)
  YYUNPUT('+')
%

+[0-9]+ ...
```

**BEGIN** is a macro which takes a name as an argument and tells **lex** to enter the start condition of that name. Until the next **BEGIN** action is executed, rules with the given start condition will be active. Rules with other start conditions will be inactive. Rules with no start conditions at all are active. To go back to the normal state where only the rules with no start conditions are active, do a "BEGIN(0)". The use of this action is explained more fully in the section on left context.

5. **Resolving Ambiguities.**

If two or more rules are matched, the action of the one which matched the most text is executed. If two or more rules matched the same amount of text, the action of the rule listed **first** in the **lex** input is executed. So if the rules are:

```
input  call act1
in    call act2
[a-z]+ call act3
```

then if the input is "in", "act2" will be called, even though both the second and the third rules matched the same text. If the input is "input", "act1" will be called (though all three rules matched, the first and third matching all five characters of the input and the second matching the first two characters of the input). If the input is "inputin", "act3" will be called, since it matched the most text. If the rules were rearranged to be:

```
[a-z]+ call act3
in    call act2
input  call act1
```

then all of the inputs "in", "input", and "inputin" would result in "act3" being called.

6. **More general input format.**

Up to now, we have used only the rules section of the input file. There are actually two more sections, delimited by "%%", which we haven't seen used
yet. The general form of a lex input file is:

```plaintext
<global definitions>
%%
<local definitions>
<rules>
%%
<user routines>
```

Any or all of the three sections may be empty. The first "%%" is mandatory. If an end of file is encountered in the second section, it is assumed that the third section is empty. Thus, the shortest legal lex input is:

```plaintext
%%
```

The scanner generated from this simply copies its input to its output (because every character will be processed with the default action of echoing unmatched text).

7. Global Definitions section.

The definitions section can contain three types of definitions: name definitions, ratfor definitions, and start condition definitions.

Name definitions are similar to the ratfor define statement. They let you set up a shorthand name for a long or frequently-used expression. The format of a name definition is simply

```plaintext
<name> <translation>
```

For example,

```plaintext
DIGIT [0-9]
LETTER [a-zA-Z]
```

To use a name definition, you must put the name inside curly braces. Here is an example of a name definition which uses the two previously defined names:

```plaintext
IDENT {LETTER}{LETTER}{DIGIT}*
```

Names may be made up of any printable characters, except that they must not begin with a digit. Name definitions must start in column 1. Note that name definitions act as though they are surrounded by parentheses. Thus

```plaintext
EXAMPLE bletch
%%
{EXAMPLE}+
```

matches one or more occurrences of the string "bletch", and does not match the string "bletc" followed by one or more h's. Name definitions are not recognized inside of double quotes or inside of character classes. Thus, the
pattern

[{DIGIT}{LETTER}]  

matches an occurrence of one of the characters '{', 'D', 'T', 'G', 'T', '}', 'L', 'E', or 'R'.

Ratfor definitions are simply any ratfor declarations which the ratfor actions in the rules section require. Note that these definitions are external to the scanner lex generates. This is not the place to declare variables needed for the actions. The definitions are used to define (or redefine) macros used by the generated scanner and its actions, and to include symbol files to be used by the entire program. You can enter ratfor definitions in either of two forms. The normal format is:

<whitespace> <code>

That is, anything that does not start in column 1 is assumed to be a ratfor definition. Ratfor definitions which must start in column 1 can be entered in the following form:

%{  
<code>  
%}

Start condition definitions have the form:

%s name ...

Start conditions are referenced in the rules section by beginning a regular expression with "name", and by the special action "BEGIN(name)". Any rule which is preceded by "<name>" will be active only at certain times. Since start conditions are a little hard to understand, they are explained in more detail in the section on left context.

Comment lines (lines beginning with a '#' in column 1) are ignored.

8. Local definitions. After the first "%%", but before any rules, any indented code is copied into the function lex generates. This code may be used to declare objects needed by the generated function but not needed elsewhere. The code is local to the generated function. As an example, suppose you wanted to count how many times the string "feeblevetzer" occurs in a file:

%%  
integer count  
count = 0  
feeblevetzer  count = count + 1
would do the trick.

9. User routines section.

This section is simply copied verbatim to the output program. Any user-written subroutines or functions referenced by the actions may be put here. The main program may also be put here, or, if the user prefers, the main program and auxiliary routines may be compiled separately and linked in. No default main program is provided. The simplest one (which can easily be included in the user routines section) is:

```
DRIVER(prog)
  integer yylex, i, dummy
  i = yylex( dummy )
  DRETURN
end
```

10. Left context sensitivity.

Sometimes it is desireable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems. The % operator, for example, is a prior context operator, recognizing immediately preceding left context, just as $ recognizes immediately following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

This section describes two means of dealing with different environments: a simple use of flags, when only a few rules change from one environment to another, and the use of start conditions on rules. In both cases, there are rules which recognize the need to change the environment in which the following input text is analyzed. These rules set up some state to reflect the change. The state may be a flag explicitly tested by the user's action code. Such a flag is the simplest way of dealing with the problem, since lex is not involved at all. It may be more convenient, however, to have lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when lex is in that start condition. The current start condition may be changed at any time.

Consider the following problem: copy the input to the output, changing the word "magic" to "first" on every line which began with the letter "a", changing "magic" to "second" on every line which began with the letter "b", and changing "magic" to "third" on every line which began with the letter "c". All
other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a flag:

```%
NOIMPLICIT
integer flag
flag = 0
%
a flag = 1; ECHO
%b flag = 2; ECHO
%c flag = 3; ECHO
@n flag = 0; ECHO
```

```magic %{
switch ( flag )
{
    case 1:
        call putlin( "first", STDOUT )
    case 2:
        call putlin( "second", STDOUT )
    case 3:
        call putlin( "third", STDOUT )
    default:
        ECHO
}
}%
```

The same problem can be handled with start conditions. Each start condition must be introduced to `lex` in the definitions section with a line reading

```%
%s name ...
```

Start conditions may be declared in any order. The conditions are referenced at the head of a rule with the `< >` brackets:

```<name>expression```

This rule is only recognized when `lex` is in the start condition `name`. To enter a start condition, use the action statement

```BEGIN(name)```

which changes the start condition to `name`. To resume the normal state,

```BEGIN(0)```

resets the initial condition of the `lex` scanner. A rule may be active in several start conditions. This is specified with the prefix:

```<name1,name2,name3>```
The same example as before can be written:

```
%s AA BB CC
%

%c ECHO; BEGIN(CC)
@n ECHO; BEGIN(0)

<AA>magic call putlin("first", STDOUT)
<BB>magic call putlin("second", STDOUT)
<CC>magic call putlin("third", STDOUT)
```

where the logic is exactly the same as in the previous method of handling the problem, but `lex` does the work rather than the user's code.

11. **Lex flags.**

`lex` recognizes the following flags:

- `-d` specifies that `lex` should put the generated scanner in *debug* mode. This means that whenever the scanner recognizes a rule in its input, a line of the form:

  ```
  --accepting rule #<n>
  ```

will be written to the error output, where `<n>` is the number of the rule which was recognized (rules are numbered in the order they are listed in the `lex` input file, starting with 1). This option is useful for determining whether the patterns you actually wrote do what you think they should do.

- `-i` specifies that the scanner should be *case-insensitive*. You may write the input rules in any case you like (and mix cases freely), and all rules will be matched regardless of the case of the input. For example, if one rule is:

  ```
  fooBar
  ```

and you specify `-i` then the strings "fooBar", "FOOBAR", "fooBar", and "FO0bAR" will all match the rule. If you don't specify `-i` then only the string "fooBar" would match the rule.

12. **Advanced Hacking.**

This section documents some of the more advanced `lex` capabilities which provide the user with more flexibility when writing a scanner.

Start conditions come in two flavors: *inclusive* and *exclusive*. *Inclusive* start conditions are declared using "%s", and *exclusive* start conditions are declared using "%x". If the pattern-matcher is in an *inclusive* start
When an unescaped double quote is seen, the start condition DQUOTE is entered (the """" is escaped in the lex rule because it is a lex meta-character). When an unescaped single quote is seen, the start condition SQUOTE is entered. The scanner stays in the start condition until the matching quote is seen. If an escape sequence of the form @@ (where 'c' is any character) is encountered while the scanner is in the DQUOTE start condition, the "@@?" rule will be matched (since it matches more text than the "<DQUOTE,SQUOTE>?" rule) and the escaped character will be returned as a "TOK_CHAR" to the routine that called the scanner. Note that if the scanner is in the SQUOTE start condition, the "@@?" rule will not be matched, because SQUOTE is an exclusive start condition. Thus these rules provide for two different types of quoted strings: one in which escapes are evaluated, and one in which they are not.

"string with escaped @" in it" will be "seen" by the routine calling the scanner as the string:

string with escaped " in it

while the string:

'\'string with escaped @' in it'
will be seen as:

```
string with escaped @
```

followed by whatever rules match the string:

```
<space> in it'
```

Another way of looking at exclusive start conditions is that they give a mechanism by which several completely separate scanners may be specified to `lex`. Each exclusive start condition corresponds to an individual scanner.

Several other flags are available to control `lex`'s processing:

- `-s` suppresses the default rule. Instead of echoing any text which doesn't match a rule, if unmatched text is encountered by the scanner it will abort with the error message `lex scanner jammed`

This option is useful for debugging scanners which are supposed to have rules to match every input string (e.g., the scanner for a compiler).

- `-t` puts `lex` in `trace` mode. For most users, only the first part of the trace information will be of interest. It is a list of each rule as `lex` parsed the rule (after macro expansions). The rules are preceded by their rule numbers. Character classes listed in the form:

```
[ <n> ]
```

for a number `<n>` were considered identical with a previous character class. To find out what the identical character class was, count each occurrence of a character class in the trace which is `not` of the form

```
[ <n> ]
```

until you reach the `<n>th` character class. (Start counting with 1 and count the first occurrence of `?` as a character class.)

The remainder of the trace will be of interest only if you are familiar with finite-automaton pattern-matching. The non-deterministic finite automaton (NFA) is dumped in the form:

```
state # <n> <char>: <first> <second> [ <accept> ]
```

where `<n>` is the number of the NFA state, `<char>` is the character which the state makes a transition on (0 is `epsilon`), `<first>` is the first transition made by the state, and `<second>` is the second transition made by the state (non-zero only for `epsilon` states). If `[@ <accept>@]` is present, it indicates the number of the rule accepted by entering the state. Following the NFA dump is a dump
of the Deterministic Finite Automaton (DFA). The dump is of the form:

\[
\text{state} \# <n>: \text{<ec>} \quad \text{<next>}
\]

where \(<n>\) is the number of the DFA state, \(<ec>\) is the equivalence class on which a transition is made, and \(<\text{next}>\) is the state to enter if the equivalence class of the input character is \(<ec>\). Intermixed with this dump will be lines of the form:

\[
\text{state} \# <n> \text{ accepts: [rule number ... ]}
\]

which lists the numbers of the rules matched upon entering state \(<n>\). Next is a list showing the equivalence class of each of the input characters. Finally, the meta-equivalence class of each equivalence class is listed.

\(-v\) specifies \textit{verbose} mode. \texttt{lex} will write a summary of its processing to the error output. A sample summary (taken from a large \texttt{lex} input) is:

\[
\text{lex usage statistics:}
\]

\[
\text{started at 22:27:53, finished at 22:28:29}
\]
\[
\text{NFA size = 772 states, DFA size = 434 states (2170 words)}
\]
\[
\text{68 rules, 3 start conditions}
\]
\[
\text{359 epsilon states, 259 double epsilon states}
\]
\[
\text{32 character classes needed 299 words of storage, 63 reused}
\]
\[
\text{14972 state/nextstate pairs created}
\]
\[
\text{454 base/def entries created}
\]
\[
\text{950 nxt/chk entries created}
\]
\[
\text{2 empty table entries}
\]
\[
\text{28 protos created}
\]
\[
\text{20 templates created, 315 uses}
\]
\[
\text{55 equivalence classes created}
\]
\[
\text{7 meta-equivalence classes created}
\]
\[
\text{27 hash collisions, 556 DFAs equal}
\]

Notes:

The times given at the top are real-times. NFA size is the number of states in the Non-Deterministic Finite Automaton. DFA size is the number of states in the Deterministic Finite Automaton. The number of integer words needed to store the DFA is given in parentheses: it is bounded by the amount of dynamic memory available to \texttt{lex}.

Ideally all of the epsilon states should be double epsilon states. There is a guaranteed overhead of one non-double state per rule and two per start condition (counting the "no start-condition" state).

A character class is "reused" when it occurs more than once in the input. The number of state/nextstate pairs is how many DFA transitions there were. base/def and nxt/chk entries refer to the scheme which is used to compress
the representation of the state/nextstate pairs. Twice the sum of the base/def and nxt/chk values is the number of integer words needed to represent the state/nextstate pairs using the compression scheme.

**lex** makes an effort to avoid empty (unused) entries in the tables it generates, though it is tuned only for large numbers of state/nextstate pairs. The number of empty entries should be a small fraction of the number of nxt/chk entries. If it is not, and the number of state/nextstate pairs is large (say > 3000), consult your **lex** guru about tuning the table compression. The number of protos measures how many different areas of redundancy **lex** found in the state/nextstate pairs. The number of templates indicates how many different "catch-all" patterns (e.g., "?*", "[a-z]+") were used to compress the representation of the state/nextstate pairs. The more templates uses the better the compression. If the number of uses is less than twice the number of templates, **lex** may be in need of tuning.

An equivalence class is formed for each group of characters which always occur together in the input. All of the information needed to run the DFA is coded in terms of equivalence classes, and not individual characters. A meta-equivalence classes is created for each group of equivalence classes which always occur together in a template. Meta-equivalence classes are used to represent templates in a compact manner.

The ratio of hash collisions to number of DFAs that were equal is a measure of the effectiveness of **lex**'s hash function. The ratio should always be quite low (the one shown is typical). If not, inform your **lex** guru so **lex** can be sped up.

There are several macros which can be redefined to customize the scanner **lex** generates. The redefinitions are normally made in the first section of a **lex** input. The macros are defined in the include file **lskdef**, and here are their functions:

- **YYBUFSIZE** is the size of the two buffers the scanner uses to hold the input characters and the state that was entered upon reading each character. It should be at least **MAXLINE** in size.
- **YYBUFMAX** sets the actual limit on how much of the two buffers the scanner will really use. Making this value lower than **YYBUFSIZE** enables you to use the end the buffer for your own use.
- **YYINIT** is executed when the scanner is first called, and whenever it encounters an EOF.
- **YYOUTPUT** is the routine used by **ECHO** to write out characters. It takes a single character as an argument.
- **YYWRAP** controls the scanner's EOF processing. The macro takes one argument. If it assigns the value 0 to the argument, the scanner will continue processing (it will be assumed that you have set things up so that it can keep calling **YYINPUT** to obtain more input). A typically redefinition of **YYWRAP**
looks like:

```c
define(YYWRAP,
   $1 = 1
   if ( getarg( 1, arg, MAXARG ) != EOF )
   {
      fd = open( arg, READ )
      if ( fd != ERR )
         $1 = 0
         call delarg( 1 )
   }
)
```

where you have declared `arg`, `fd`, `getarg()` and `open()` in the local declarations part of the `lex` input.

`YYEOLFACTION` is executed when the scanner encounters an `EOF` and `YYWRAP` returns a non-zero value.

`YYDEFLACTION` is executed when none of the rules match the input. The "matched text" is the first character of the unmatched input.

`YYDECL` is used to declare the scanning routine. For example, if you wanted the scanning routine to be called `lexscan`, to take two arguments, one integer and one character, and to return a character value, you could use:

```c
define(YYDECL,
   character function lexscan( iarg, carg )
   integer iarg
   character carg
)
```

In comparison, the default declaration is:

```c
define(YYDECL,
   integer function yylex( yyval )
   integer yyval
)
```


Here are the patterns which are common to all tools which use regular
expressions:

c  the character "c".
[...]  character class--any one of these characters.
[x-z]  a character class specified as a range.
[!...]  negated character class--any character but these.
?  any character but NEWLINE.
e*  0, 1, 2, ... instances of e.
%e  an e at the beginning of a line.
e$  an e at the end of a line.
@c  a "c", even if c is an operator.
@xxx  any valid escape sequence (see esc(2))

Here are the operators which are unique to lex:

"xyz"  xyz, even if it contains operators.
e+  1, 2, 3, ... instances of e.
e|f  an e or an f.
(e)  an e. Used to group operators, like in arithmetic.
e/f  an e, but only if followed by f.
{xx}  the translation of xx from the definitions section.
{m..n}  m through n occurrences of e.
{m..}  m or more occurrences of e.
{m}  exactly m occurrences of e.
e  an optional e
<a>e  an e when the scanner is in start condition a.

14. Example.

The following lex program processes a simple calculator language. The program reads in real numbers and performs operations upon them. The numbers are of the form

    ddd[[ddd]]

where ddd indicates one or more digits and brackets indicate optional elements. The operators are the standard four arithmetic operators. To keep the example simple, all operators have the same precedence. When the program encounters an '=' it writes out the result of evaluating the current expression. The lex part of the program does several different tasks: it splits up the interesting parts of the input into different tokens so the main program (which does the actual parsing of the expressions) needs only deal with high-level objects (of which there are three: numbers, arithmetic operators, and the '=' operator); it weeds whitespace out of the input so the parser doesn't have to deal with it; and it detects and recovers from the one error it can easily detect--garbage in the input.
# sample lex program which parses simple arithmetic expressions

# used to return a real value from the scanner
define(REAL_VALUE_COMMON,
    real rval
        common /example/ rval
    )

define(REAL_NUMBER,1)
define(OPER,2)
define(PRINT_VALUE,3)

DIGIT [0-9]

# Start condition we enter when we want to discard the current input line. This happens when we run across something which we don't recognize.
%s EATLINE

%%

NOIMPLICIT
character num(MAXLINE)
integer idx
real ctor

REAL_VALUE_COMMON

<EATLINE>??@n  BEGIN(0)

{DIGIT}+( {DIGIT}*) %
    YYGETTEXT(num)
    idx = 1
    rval = ctor( num, idx )
    return( REAL_NUMBER )
%

[-+*/]       yyval = YYFIRSTCHAR; return( OPER )
        return( PRINT_VALUE )

[ @t@n]+     # eat whitespace
?
%
    call remark( "unrecognized character, line discarded" )
    BEGIN(EATLINE)
%

%%

DRIVER(main)
NOIMPLICIT

integer yylex, yytok, yyval, op
real expr

REAL_VALUE_COMMON

- 19 -
yytok = yylex( yyval )
while ( yytok != EOF )
{
    if ( yytok == REAL_NUMBER )
    {
        expr = rval
        yytok = yylex( yyval )
        while ( yytok == OPER )
        {
            op = yyval
            yytok = yylex( yyval )
            if ( yytok == REAL_NUMBER )
            {
                switch ( op )
                {
                    case '+':
                        expr = expr + rval
                    case '*':
                        expr = expr * rval
                    case '-':
                        expr = expr - rval
                    case '/':
                        expr = expr / rval
                    }
                }
                yytok = yylex( yyval )
            }
        else
        {
            call remark( "right-hand operand missing" )
            yytok = yylex( yyval ) # clear out current token
            next 2 # start processing another expression
        }
    }
    else
    {
        call printf( "%g\n", expr )
    }
}
if ( yytok == PRINT_VALUE )
call printf( "%g\n", expr )
else
    call remark( "'=' expected" )
else
    call remark( "left-hand operand missing" )

yytok = yylex( yyval )
END

RETURN
Note:

(1) The use of the start condition to recover from an error.

(2) How the order of the rules (in particular, the rule prefixed with the start condition coming first and the '?' rule coming last) is used to make sure that input which matches more than one of the patterns results in the correct action being executed.

(3) The '-' operator must be listed as either the first or the last member of the character class which is recognizes operators. Otherwise it would be interpreted as specifying a range of characters.

(4) How the work is divided between the scanner and the parser. It would be difficult to write this program using only lex actions.

FILES
+INCL/lskdef definitions for generated scanner

SEE ALSO

AUTHOR
Vern Paxson. Evolved from an original implementation by Jef Poskanzer, with the help of many ideas from Van Jacobson. Parts of this document were taken from the Unix Lex document. The remainder was written by Jef Poskanzer and Vern Paxson.

BUGS/DEFICIENCIES
See the lex(tool) manual entry for a list of the bugs and deficiencies.
NAME
Msgtut - explain how to use msg.

SYNOPSIS
msgtut

DESCRIPTION

1. Introduction

_Msg_ provides a friendly environment for sending and receiving mail. It offers the user simple editing capabilities to ease the composition of outgoing messages, as well as providing the ability to read, write and modify files which have the message file format.

The _msg_ system accepts incoming messages for you from other people and collects them in a file, called _mymail_. When you login, the system notifies you if there are any messages waiting in your _mymail_ file. When you read your mail, it reads your _mymail_ file and separates that file into the individual messages that have been sent to you. You can then read, reply to, delete, forward to or save these messages. _Mbox_ is the file where the messages from _mymail_ are saved by default.

Finally, _msg_ is able to send and receive messages across networks such as the DECENT, ARPANET, and UUCP.

2. Common Usage

The _msg_ has two distinct usages, according to whether one wants to send or receive mail. All the 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:

```plaintext
msg [-p[n]] [filename]
```

If no filename is specified, _msg_ defaults to read 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. Then it will type a prompt "<-", a left arrow, and await your command.

There are only five types of input expected by _msg_:

1. a _msg_ command character.
2. a message sequence specification.
3. a filename.
Msgtut(tutorial)

April 11, 1985

Msgtut(tutorial)

4. a confirmation character <SP>.
5. an output continuation character <SP>.
Whenever msg prompts for input, typing <ESC> causes the current command
to be aborted, and the user is returned to command level. The conventions
which are used in the command descriptions (e.g. filename, message
sequence specification) are listed in Section 6.
The banner that msg greets the user with is :
Software Tools MSG system
type ? for help
type # for news
type % for intro

<-

2.1. Sending mail
When the user types an "s" (send mail) command, msg will prompt for
input of. a list of "To:" addresses and the body of the message. The list of
addresses should consist of user-names separated by commas. If the address
list is longer then one line, type an atsign, "@", at the end of the line for continuation.
After inputing the addresses, the user will be put directly into the message
input mode. When you reach the end of the message. type a "." character
alone on the line. This will cause msg to send the message to the delivery systerri and go back to the msg command level. If you change your mind about
sending the message, you can type ""'q" as the only character on the line to
quit and return to msg command level.
An example of sending mail:

<- s end mail [type SPACE to confirm] <SP>
To: jane. leres. van, tab, barale, marshall, jpl@

T_chang
Subject: testing
Message (,"'?' for help, . .' to end, ''''q' to quit)

Test message for msg system
Submit mail to MAILER

<-

-2-


2.2. Reading mail

In msg, to read the current message press carriage return <CR>. The "b" (backup) and "n" (next) commands allow you go back and forth in the mail messages. To look at a specific message, use the "t" (type) or "g" (goto) command.

For example:

<- <CR> next message is:

(message 1, 85 characters)
Date: Thur, 15 Dec 83 09:02:52 PST
From: sysmgr <system manager) @ rtsgvx
Message-Id: <831215090252.001rtsgvx>
Subject: system shotdown
To: all

System will be down tonight at 10:00pm for hardware maintenance.
<- 

If, after reading a message, you wish to immediately send a reply, you can do so with the "a" (answer) command, or forward the message to the others with the "f" (forward) command.

<- answer message number: <CR> current message
Message ('~?' for help, '.' to end, '~q' to quit)
Thanks for your notice.

Submit Mail to MAILER
<- 

Normally, each message you receive is saved in the file mbox in your home directory at the time you exit the msg. If you do not wish to save a message, you can delete it by using the "d" (delete) command to avoid saving it in mbox.

<- delete (message sequence) 1-4 <CR> <- 

This deletes the first four messages from the message file.

If you accidentally delete a message, you have the command "u" (undelete) to recover your deleted messages.

There are some additional commands which are used for reading mail, such as "p" (put message into a file), "o" (overwrite the file), "m" (move message to another mail file and delete it) and "h" (list the headers of the messages in
the message file).

For example:

\[
\text{<- put (message sequence) 1 <CR> into file name: junk <-}
\]

Or, if you want to put the message into a file and then delete it, you can use the "m" (move) command.

\[
\text{<- move (message sequence) 1 <CR> into file name: junk <-}
\]

More information about \textit{msg} commands are listed in Section 6.

2.3. Exit from \textit{MSG}

In \textit{msg}, you have both "e" (exit) and "q" (quit) commands to exit the \textit{msg} session. When you use "e" to exit \textit{msg}, the file \textit{mymail} will be appended to the file \textit{mbox}, and all the messages in \textit{mymail} will be deleted. If you use "q" to exit \textit{msg}, any mail message which you have marked for deletion is not deleted and all the messages are kept intact in the \textit{mymail} file and will be relisted the next time you get into \textit{msg}.

3. More about sending mail

3.1. Tilde escapes

While typing in a message to be sent to others, it is often useful to be able to invoke the text editor on the partial message, print the message, execute a shell command, or do some other auxiliary function. The send mail facility of \textit{msg} provides these capabilities through \textit{tilde escapes}.

Tilde escape commands have a tilde "~" at the beginning of the line, followed by a single character which indicates the function to be performed.

Valid commands are:

\[
\begin{align*}
~? & \quad \text{- to display this help list} \\
~! \text{ command} & \quad \text{- execute shell command} \\
~b \text{ address(es)} & \quad \text{- add address(es) to Bcc: field} \\
~c \text{ address(es)} & \quad \text{- add address(es) to cc: field} \\
~e \text{ [editor]} & \quad \text{- edit the message using [editor]} \\
~p & \quad \text{- print the message so far}
\end{align*}
\]
~r file     - read file into message
~s text string - add text string to Subject: field
~t address(es) - add address(es) to To: field
~w file     - write message on file
~~          - begin a line with a '~'

Any other input results in an error message.

You could use the pathname "'/user/bin/edt" or the vms file specification "drc2:[user.bin]edt" to specify your editor. i.e.

~e /usr/bin/edt or ~e drc2:[usr.bin]edt

If an editor name is not specified in the "~e" command, then "/usr/bin/ed"
is spawned. If the person(s) you are sending mail to are currently logged in, they will receive the message

Software Tools mail has arrived from '<your name>'

3.2. Addresses

You can send mail to either user names or alias names. The valid form of the addresses are:

    name [ at host]
    name [@host]

'name' can be any of the user names displayed by the "users" tool, as well as, any of the alias names found in '~/malias' (the mail alias file in your home directory). When host name is specified, the mail delivery system attempts to send the mail to the host, with the mail eventually being delivered to 'name'.

Comments may be embedded in an address by surrounding the comment with parentheses, as in

    van (Van Jacobson) at rtsvgvx
    rtsg (RTSG colman) @lblg

Aliases:

The format of aliases in '~/malias' are:

    alias_name: address[,address]...;

If the alias occupies more than one line, a trailing comma (';') indicates that the alias is continued onto the next line, as in the address lists given in response to the To:, cc: and bcc: prompts. For example:
tools: ed, jane, marshall, lerés, chang, tab, van,
    vern, xero,jpl;

3.3. Network operation

To determine the valid users on your machine, type "users -v". To deter­
mine the valid users on a particular host (<host>), execute the follow­ing
command from the shell:

    % echo "users -v" | mail "rmtxeq at <host>"

Of course, if it is necessary for the user to route the message to <host>, the
appropriate ‘via host[ via host]...’ information can be added to the above
address. You will receive the output of the users command run on <host> as
a return message.

4. Set and Ignore Options

Msg always looks for two files when it is invoked. It first reads a system
wide file "~msg/msg.cf" (drc2:[msg/msg.cf]), then a user specific file,
"~/msg.cf", i.e: drc0:[randy]msg.cf, which is found in the user's home direc­
tory. The system-wide file is maintained by the system administrator and
contains set options that are applicable to all users of the system. The
"~/msg.cf" file is usually used by each user to set desired options or ignore
message header fields.

It is possible to set several options in the same set command. The options
can be binary, in which case they can be switched on or off, or valued. To
set a binary option on, do

    set option

To give the valued option a value, do

    set option=value

Space is not allowed among option, ",=" and value.

The possible set options are listed as follow:

binary options:

    ask    prompt for "Subject:" when sending mail.
    askcc  - prompt for "Cc:" when sending mail.
    askbcc - prompt for "Bcc:" when sending mail.
quiet - suppress printing the banner when invoking "msg".

valued options:

---

escape - redefine the escape character instead of "~".
editor - specify pathname of editor for "~e".
shell - specify pathname of shell for "!".

Using *ignore* command, you can add the list of header fields named to the ignore list. Header fields in the ignore list are not printed on your terminal when you print a message. This allows you to suppress printing of certain machine-generated header field, such as *Message-ID* which is not usually of interest. The "t" commands can be used to print a message in its entirety, including ignore fields.

The fields that can be ignored are listed as follow:

to : To field
bcc : Bcc field
cc : Cc field
recv : Received From field
date : Date field
path : Return Path field
subj : Subject field
msgid : Message-ID field
status : Status field
reply : In-Reply-to field

For example, the following two lines in the msg.cf

```
set ask askcc escape=?
ignore status msgid
```

would cause sending mail facility to prompt the user for a Subject, a Cc (carbon copy) address list, all the *tilde escape* commands will start with the character "?" instead of "~", and message header fields "Status" and "Message-ID" will not show on the screen when reading the message.

5. Command Line Option

This section describes command line options for *msg* and what they are used for.

*-f* is used to change the number of fill characters written to the terminal whenever a carriage-return, line-feed pair is written.
-p can be used to change the page size (number of lines per screenful) for 'msg' to use when displaying long output. The default is 22 lines.

6. List of Commands

This section gives a summary of the msg commands and the conventions which are used in the command descriptions.

6.1. Msg Commands

a answer message number: <message number>

This command causes msg to spawn a sub-process stmail, with the To field being the sender of the indicated message, and 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.

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.

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

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.

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.
f forward message number: <NUMBER>

This command causes the \textit{stmail} 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 the \textit{stmail}.


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 $1 \leq n \leq \text{NMSGS}$
2. f for the first message (message number 1)
3. l for the last message
4. <CR> for the current message number (a noop)

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:

<status> <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.

The possible status of the message are listed as follows:
"N" - new message.
"D" - deleted message.
"A" - answered message.

i ignore header fields: (field)

This command causes the ignored header fields not to show on the screen when reading the message.

The fields that can be ignored are listed as follow:
\begin{itemize}
  \item to: To field
  \item cc : Cc field
  \item bcc: Bcc field
  \item rec: Received From field
  \item date : Date field
  \item path : Return Path field
\end{itemize}
subj: Subject field  msgid: Message-ID field
status: Status field  reply: In-Reply-to field

You can ignore several fields within one ignore command, each field should be separated by a space. The command "t" (type) is the only command that allows you to display those ignored fields when reading the message.

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.

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.

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.

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.

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, renumbering the messages.

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.

q  quit [type SPACE to confirm] This command allows the user to leave MSG without modifying the current message file.

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.

s  send mail [type SPACE to confirm]

This command causes sending mail facility to be spawned as a subprocess, allowing the user to send a message without leaving msg; when it exits, msg regains control.

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. Also, it displays the whole messages including all ignored header fields.

u  undelete (message sequence) <MSG-SEQUENCE>

This command undoes the actions of the Delete command.

w  what command information - type command character: <COMMAND-CHARACTER>

?  This command will display the summary of the msg commands as follows:

a[nswer] message
b[ackup] to previous message and type it
current] message number and file
d[lete] message(s)
e[xit] and update old file
f[oward] message
g[oto] message specified and print it
h[eaders] print headers of message(s)
i[gnore] header field(s)
j[ump] into shell, return by typing logout to shell
KEY encryption-key *** UNIMPLEMENTED ***

LIST message(s) in print format on file

MOVE message(s) to a file and mark them deleted

NEXT message is typed

DELETE current file and re-read

PUT copies of message(s) in another mail file

QUIT leave MSG without updating current file

READ in another mail file

SEND mail to send a message (and return to MSG)

TYPE message(s) on standard output

UNDELETE message(s)

WHAT command information to display

#NEWS print MSG news

?HELP print this list

%INTRO type an introduction to MSG (for new users)

For more information, use the WHAT command.

# news command will inform the user of recent changes of the system.

% will display the very brief introduction of the msg for the first time user.

6.2 Conventions of the Command Descriptions

The following conventions are used in the command descriptions:

<FILE-NAME>

This stands for any valid file specification on your system.

(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.
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 <CR>.
4. Special types of message sequences, which are determined by the first character typed in response to the (message
sequence) prompt.

<table>
<thead>
<tr>
<th>character typed</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;CR&gt;</td>
<td>The relevant process is performed on the current message</td>
</tr>
<tr>
<td>a</td>
<td>The string &quot;all messages&quot; is displayed and the relevant action is taken on all messages</td>
</tr>
<tr>
<td>d</td>
<td>The string &quot;deleted messages&quot; is displayed and the appropriate action is taken on those messages currently marked as deleted</td>
</tr>
<tr>
<td>f</td>
<td>The string &quot;from string: &quot; 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.</td>
</tr>
<tr>
<td>s</td>
<td>The string &quot;subject string: &quot; 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.</td>
</tr>
<tr>
<td>u</td>
<td>The string &quot;undeleted messages&quot; is displayed and the relevant action is taken on those messages currently not deleted</td>
</tr>
</tbody>
</table>

The strings required for the from and subject search are the same regular expressions used by the editor. The string must be terminated by a "<CR>". If a bare <CR> is typed, no searching is done and the command is terminated.

**FILES**

- mymail - messages sent using the 'stmail' utility reside here.
- mbox - default file for saving messages from mymail.
- one scratch file is used by 'msg'.

**SEE ALSO**

- mail - the utility for sending mail to local users
- stmail - the utility for sending mail to other users
- msplit - the utility for salvaging message files
AUTHOR

Whei-Ling Chang. Section 6 was written by Joe Sventek.
NAME
ratfor - a primer

SYNOPSIS
rat4 [files ...] > outfile

DESCRIPTION

INTRODUCTION

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" for conditionals
"switch" for a type of multiple conditional
"while", "for", and "repeat"-"until" for looping
"do" for ratfor style do loops
"break" and "next" for controlling loop exits
"return" for ratfor style return statement
statement grouping with braces

The cosmetic aspects of ratfor have been designed to make it concise, easier to use, and reasonably pleasing to the eye:

free form input
unobtrusive comment convention
translation of >, <=, etc. into .GT., .LE., etc.
quoted character strings
"string" statement for setting up character strings
integers in bases other than ten
"define" statement for symbolic constants
"include" statement for including source files

Ratfor is implemented as a 2-pass preprocessor. The first pass, ratp1, translates the above features into fortran. The second pass, ratp2, orders the translated code. The output 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--\{\}--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 "else" statement to handle the construction "if a condition is true, do this thing, otherwise do that thing". The syntax is

```
if (legal ratfor condition)
  statement(s)
else
  statement(s)
```

where the else part is optional. The "legal ratfor condition" is anything that can legally go into a ratfor or 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
```
SWITCH, CASE, and DEFAULT

The "switch" statement will execute one of a number of sections of code depending on the result of some calculation. The syntax is:

```
switch (expression)
{
    case range, range, ...:
        statement(s)
    case range, range, ...:
        statement(s)
    ...
    default:
        statement(s)
}
```

The range specified in the "case" statements may be of the form "constant" or "constant-constant". If the first form is used, the range applies only to the constant specified. In the second form, the range applies to anything between the first constant and the second constant, inclusive. The dash should not be confused with subtraction.

In the switch statement, "expression" is first evaluated. If it is equal to any of the constants specified in the first case statement, the statements following the case statement are executed up to the next case statement and control is returned to the statement following the switch statement. Similarly, if "expression" is equal to any of the constants specified in the second case statement, those statements are executed and so on. If "expression" is not equal to any of the constants specified in any of the case statements, the commands following "default" are executed. If there is no "default" statement in such a case, nothing is done. The same constant may not be specified in two different "case" statements. Also note that the statements following "case" or "default" do not need to be within braces. For example:

```
switch (getc (c))
{
    case 'a' - 'z':
        call remark ("I read a letter.")
        alpha = alpha + 1
    case '+', '-':
        call remark ("I read a sign.")
        sign = sign + 1
    default:
```
This functions the same way as:

```c
    c = getc (c)
    if ( c >= 'a' & c <= 'z' )
        { call remark ("I read a letter.")
          alpha = alpha + 1
        }
    else if ( c == '+' | c == '-' )
        { call remark ("I read a sign.")
          sign = sign + 1
        }
    else
        call remark ("Illegal character.")
```
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 ratfor condition)
  statement(s)
```

As with the if, "legal ratfor condition" is something that can go into a ratfor or 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)
}
```
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_clause; condition; increment_clause )
statement(s)
```

where "init_clause" is one or more ratfor statements separated with commas, which get done once before the loop begins. "Increment_clause" is one or more ratfor statements separated by commas, which get 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_clause, condition, and increment_clause 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, m=1; j>0; j=j-1, m=m+1 )
    {
      call putc (line(j))
      temp(m) = line(j)
    }
  }
  call close (int)
}
```

The above code gets file names from the command line. For each file specified, the lines are read one at a time from the file. The characters in each line are output backwards onto the standard output file. The characters are also copied backwards into a "temp"orary array. (The "!=" means .NE.)
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 )
```
Ratfor implements the ratfor do statement as well as the fortran one. The syntax is:

```ratfor
    do range
        statement(s)
    
    "Range" may be any valid fortran range. For example:

        do i = 1, 100, 2
            {  
                call remark ( "The next odd number: " )
                call putdec (i, 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", or "repeat"). 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:

```ratfor
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:

```ratfor
for (i=1; i<10; i=i+1)
{
  if ( array(i) == ' ' ) next
}
```

The forms "break n" and "next n" may be used to break or reiterate over multiple loops. "Break n", for example, will cause the next n loops to be exited, and "next n" will exit the next n - 1 loops and then continue to the next iteration of the loop after that. "Break 1" is equivalent to "break", and "next 1" is equivalent to "next".
RETURN

Ratfor implements its own "return" statement as well as the fortran one. The ratfor return statement with a value is only meaningful in a function. Like the fortran return statement, it will return control to the calling program, but it will also set the function value to the value specified. The syntax is:

    return (value)

For example, if this statement were in function sqrt, it would set sqrt to "value", and return.
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 remark("ERROR: maximum size exceeded")
    err = MAX_EXCEEDED
    return (ERR)
}
```

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

Ratfor 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 (as in a for statement); 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. Multiple statements 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 otherwise.

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 putting a ":\%" as the first character on the line. The percent will be removed and the line shifted one position to the left, but otherwise the statement will remain unchanged. This is a convenient way to pass regular fortran or assembly code through the ratfor processor. Parts of a line can also be passed through the processor unchanged. Sequences of characters can be surrounded with the characters ":\%(" and ":\%)", thus avoiding processing. The surrounding characters will be removed and the character sequence will be output without change. For example,

    call hollstr( 19, ":\%(19H1234567890123456789%)\) )

The hollerith string 19h1234567890123456789 will not be processed by ratfor (although ratfor would have left it unchanged anyway).

Sequences of characters can be passed through the processor, thus avoiding processing, by surrounding them with the tokens ":\%(" and ":\%)". The surrounding tokens will be removed and the character sequence will be output without change.
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,

```fortran
integer function dummy(x)
  integer x

  #simply return the dummy parameter
  dummy = x

  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:

< .lt.
<= .le.
== .eq.
!= .ne.
>= .ge.
> .gt.
| .or.
& .and.
! .not. ~ .not. ~ .not.

For example,

```fortran
for ( i=1; i <= 5; i=i+1 )
   ...

if ( j != 100 )
   ...
```
THE STRING STATEMENT

To make character strings easier to set up, ratfor implements a "string" statement. This is not an executable statement, but acts like a type statement such as "character" or "integer", and must be placed in a place appropriate to this type of statement. The syntax is:

\[
\text{string name } "\text{value}" \quad \text{or} \quad \text{string name(size) } "\text{value}"
\]

This statement causes ratfor to set up a character string by the name "name" that is large enough to hold "size" characters, and initializes it to contain the string "value", left justified, with a trailing EOS. If the "(size)" parameter is not given, rat4 makes the string only large enough to contain the string "value" and its EOS. The size specified should not be smaller than the number of characters required to hold the initial value. For example the user could say:

\[
\text{string errmsg } "\text{fatal error in subroutine x}" \\
\]

Then the string could be used as follows:

\[
\text{call error (errmsg)}
\]

Note that the ratfor string statement does NOT generate hollerith strings, but character strings, so its use is not system dependent. In-line strings can be continued across line boundaries by ending the line to be continued with an underline character. The underscore is not included as part of the literal string. Leading blanks and tabs on the next line are ignored. If a double-quote (" ) is to be embedded in a string it must be escaped. For example,

\[
"\text{a quote (@" in a string}" \\
\]

In addition the normal escape sequences are supported in ratfor (see esc(lib)).
QUOTED CHARACTER STRINGS

In-line quoted strings consist of text enclosed in double quotes outside the scope of a string declaration. These strings are converted to character array declarations with data statements. The array will be declared long enough to hold the characters in the string plus an EOS character. The 2nd pass of the ratfor processor, ratp2, takes care of ordering these declaration statements to be ANSI-66 compliant. For instance, on a machine which had the ratfor character type defined to "logical*1"

```plaintext
call remark ("Error detected")
```

would translate to

```plaintext
logical*1 st001z(15)
data st001z(1)/69/,st001z(2)/114/,st001z(3)/114/,st001z(4)/111/, st001z(5)/114/,st001z(6)/32/,st001z(7)/100/,st001z(8)/101/,st001z(9)/116/,st001z(10)/101/,st001z(11)/99/,st001z(12)/116/,st001z(13)/ st001z(14)/100/,st001z(15)/0/
call error(st001z)
```
CHARACTER LITERALS

Character constants can be used anywhere that a character variable is allowed. Characters constants are enclosed in single quotes, 'c'. They are converted by the ratfor processor to the decimal integer representation of that character in the ASCII character set. For example:

```ratfor
call putc( '@n' )  #output a newline
call putc( '3' )   #output the number 3
```

The normal escape characters are allowed as character constants, see the manual for the function "esc" for more information on legal escape characters.
INTEGERS IN BASES OTHER THAN TEN

Ratfor allows the user to specify integer constants in bases other than ten. Writing:

n%dddd...

specifies that "dddd..." should be interpreted as a number in base n, where n is given in decimal. For bases greater than ten, letters are used for digits higher than nine. For example:

2%110010 = 50 (base ten)
3%12021 = 143 (base ten)
5%423 = 113 (base ten)
7%123 = 66 (base ten)
8%77 = 63 (base ten)
16%2ff = 767 (base ten)
DEFINE

Any collection of letters, digits, periods, and underlines can be defined as a macro name, as long as it starts with a letter; thereafter, whenever that exact name occurs in the input (delimited by non-alphanumerics) it is replaced by the rest of the definition line (comments are stripped off). "Define" is used to make these symbolic parameters:

\[
\begin{align*}
\text{define} & (\text{ROW}, 10) \\
\text{define} & (\text{COLUMN}, 25) \\
\text{dimension array} & (\text{ROW}, \text{COLUMN})
\end{align*}
\]

Define (END_OF_FILE, EOF)

if \ (\text{getlin}(\text{line}, \text{int}) == \text{END_OF_FILE})

Definitions may be included anywhere in the code, as long as they appear before the defined name occurs. The name and its definition should not exceed 200 characters. Upper and lower cases ARE significant (thus EOF is not the same as eof).

Macros may also take arguments. In the macro definition, these arguments are represented by \$<\text{number}>. For example, \$1 would be the first argument and \$2 would be the second. In the macro call, the arguments are given following the macro name in parentheses. For example the user could make the following definition:

\[
\begin{align*}
\text{define} & (\text{copen}, \\
\text{\$3} & = \text{open}\ (\$1, \$2) \\
\text{if} & \ (\$3 == \text{ERR}) \\
\text{call} & \ \text{error} \ (\text{"Can't open file."})
\end{align*}
\]

This could be called as:

\[
\text{copen} (\text{"myfile"}, \text{READ}, \text{fd})
\]

and it would expand as:

\[
\begin{align*}
\text{fd} & = \text{open}\ (\text{"myfile"}, \text{READ}) \\
\text{if} & \ (\text{fd} == \text{ERR}) \\
\text{call} & \ \text{error} \ (\text{"Can't open file."})
\end{align*}
\]

Note that with "define", no part of the macro name or definition is expanded
until the macro is called, unlike macro evaluation in the "macro" program. However, ratfor does have an "mdefine" (macro-define) which expands macro names and definitions as it interprets them, like the "macro" program. The syntax of "mdefine" differs from the macro tool in that the tokens "$(" and ")" are used to indicate deferred evaluation, rather than the "[ " and "]" used by the macro tool.

In addition to define, six other built-in macros are implemented:

arith(x,op,y)

Arith performs arithmetic operations on integers. It performs the operation given by op (which may be +, -, *, **, or /) on the two arguments x and y, and returns the result.

incr(x)

Incr converts the string x to a number, adds one to it, and returns the result as a character string.

ifelse(a,b,x,y)

Ifelse compares a and b as character strings. If they are equal, ratfor compiles the code given by x. Otherwise, ratfor compiles the code given by y.

substr(s,i,n)

Substr extracts n characters of s starting at the i’th character and returns the result. If n is not present or is too big, substr returns the rest of the string, and if i is not within the string, it returns the null string.

lentok(str)

Lentok returns the length of the character string str (number of characters in str).

undefine(name)

Undefine undefines the macro name ‘name’. This means that ‘name’ will no longer be translated into a macro definition by ratfor.

Two important built-in macro definitions are DRIVER and DRETURN. These macros are used to call the necessary Software Tools initialization and clean-up routines for programs. DRIVER replaces the customary "program"
statement and takes an optional program name as it's only argument. DRETURN should be called at the end of a program. It takes an optional status argument, OK or ERR. If the status argument is omitted, it is defaulted to the value OK. This status is used by spawn to return a status to the program which did the spawning (i.e. the parent, usually the shell). If you are writing a program which does not use the Software Tools IO routines, it is not necessary to use DRIVER and DRETURN. Here is an example of a typical Software Tools program body:

```
DRIVER(main)
   call dowork(sts) #dowork sets sts
       #to ERR or OK
DRETURN(sts)
end
```

In addition, there are macros for conditionally compiling code depending on whether a macro has been defined or not:

```
ifdef(macro)
   .
   .
elsedef
   .
enddef
```

compiles the statements between ifdef and elsesdef if "macro" is defined. Otherwise the statements between elsesdef and enddef are compiled. The elsesdef can be omitted. The conditional bodies may be nested, up to 10 levels deep. Similarly,

```
ifndef(macro)
   .
enddef
```

compiles the code specified if "macro" is not defined.
INCLUDE

Files may be inserted into the input stream via the "include" command. The statement

    include filename

inserts the file found on input file "filename" into the ratfor input in place of the include statement. This is especially useful in inserting common blocks. For example,

    integer function exampl (x)
        include comblk
        exampl = x + z
        return
    end

might translate into

    integer function exampl (x)
        common /comblk/ q, r, z
        exampl = x + z
        return
    end

The actual location of the include file (in this case 'comblk') is found with the include search path. (See 'man paths' for more details.) If the include file name contains any non-alphanumeric characters it should be enclosed in double quotes:

    include "lex.c"

The include file "ratdef" is automatically included by the ratfor processor. This includes such definitions as EOF, READ, WRITE, OK, ERR, etc. If you do not wish this include file to be included you should use the -n flag with ratpl (see man ratpl). NOTE: if you use the -n option, i.e. don't include the ratdef file, you will not be able to use the string statement.
STATEMENT ORDERING

The reordering of the translated ratfor code is done in the second pass of the ratfor processor, ratp2. Ratp2 normally orders declaratory statements such as 'implicit', 'integer', 'common', 'equivalence', and 'data' in the order:

1. implicit
2. dimension, integer, real, double precision, complex, logical, external
3. common
4. equivalence
5. data

This statement ordering is done at the beginning of any main program, function, subroutine, or block data routine.

The 'beginning of the routine' is after the routine begins until some code generating statement other than implicit, ..., data.

Main programs are recognized by being at the beginning of a file, after an 'end' statement, or after a 'program' or DRIVER statement. Functions follow a 'function', 'integer [* n] function', ..., or 'logical [* n] function' statement, subroutines follow a 'subroutine' statement, and block data routines follow a 'block data' statement.

The use of variables with the names of the declaratory statements is strongly discouraged. Rat4 is not very smart about recognizing declaratory statements, but will generally compile the correct code. An exception is something like

    subroutine example
    ... declaratory statements ...
    equivalence ( a, b, c )
    data d / 3 /
    integer = 5

In this case, the statement 'integer = 5' will be placed before the data and equivalence statements, which will cause a fortran compilation error.

The user may inhibit statement ordering by invoking the 1st pass only, see man ratp1 for details.
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 does not know any 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.

Ratfor is a 2-pass processor. The passes can be invoked separately (see the manuals for ratp1 and ratp2), or together by just invoking "rat4".

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

### main - main calling routine

DRIVER( main )

    call count

DRETURN
end

### count - counts lines in files

subroutine count

    integer getarg
    character file ( FILENAMESIZE )
    integer getlin
    character line ( MAXLINE )
    integer open
    integer fd
    integer i, linect

    # the file "comblk" contains a common block which
    # defines the variable "linect"
    include comblk

    linect = 0

    # loop through the list of files
    for ( i = 1; getarg( i, file, FILENAMESIZE ) != EOF;
        i = i + 1 )
    {

        # open the file
        fd = open ( file, READ )

        for ( getlin = 0; getlin != EOF; getlin = getlin + 1 )
            linect = linect + getlin
        end
        close ( file, fd )
    end

end
if ( fd == ERR )
{
    # this file could not be located
    call remark ( "can't open file" )
}
else
{
    # read lines from the file
    while ( getlin ( line, fd ) != EOF )
        linect = linect + 1

    # close the file
    call close ( fd )
}

return
end
SEE ALSO
rat4(tool), ratp1(tool), ratp2(tool), esc(lib), sh(tool)

Also see:

1) Kernighan, Brian W., "Ratfor--a Preprocessor for a Rational Fortran". Bell Laboratories publication. Also available from the UC Berkeley Computer Science library.


3) The rat4 user and design documents; the rc user document.

4) The Unix command "rc" in the Unix Manual (rc(1))
NAME
redirect - (use of redirection symbols)

SYNOPSIS
(see below for usage)

DESCRIPTION

1. Introduction

Each of the Software Tools has a standard input and a standard output. The
default standard input and standard output is the terminal. For example,
the 'cat' command simply copies the input stream to the output stream, and
since the defaults are the terminal, the command:

    cat

will copy data from the terminal to the terminal.

Some commands do not have any data input, for example, the 'date' com­
mand. So, the command:

    date

will write the date on to the standard output stream, the terminal. For
example:

    100ct81 15:30:43

The convention of defining a standard input and standard output applies to
all the Tools that have input or output.

However, reading and writing only from and to the terminal is not very use­
ful. To be able to do anything serious we need to be able to access files, and
to change the default standard inputs and outputs to specified files. (For
information on files, see .....).

2. Redirecting Standard Input and Output

Redirection of input and output is simply the notion that the standard input
stream, or the standard output stream may have their defaults overridden.
When a Tools user says that the output of a program was redirected to the
file 'fred', he means that the default standard output stream was changed to
be the file 'fred'.

To redirect the output of a tool to a file, a greater than sign, '>', is used, fol­
lowed by the name of the file to which output is to be redirected. If we want
to put the output of the 'date' tool on a file 'fred' we say:
The tool 'cat' reads from standard input (default: the terminal) and writes to standard output (default: the terminal). To redirect standard input, a less than sign, '<' is used. So to read the file 'fred' and put the output on the terminal, we can say:

    cat <fred

Here the standard input stream was redirected, and the standard output stream was left as the default. Input and output redirection can be used together. To copy the file 'fred' to the file 'joe' the command:

    cat <fred >joe

can be used. In this case, standard input has been redirected to read from a file 'fred'; and standard output has been redirected to the file 'joe'.

This convention is used throughout the tools. The tool 'ch' changes one line to another. It has a standard input, and a standard output, and its parameters are the strings to be used. So, it is possible to say:

    ch < fred ':' '/' > joe

which will read the file 'fred' and change every occurrence of a colon, ':', to a slash '/'. The output of this operation is redirected to the file 'joe'.

There is one, and only one, standard input defined for each tool. Similarly, there is just one standard output file. However, quite often we want to do operations on several input files. For example, we may want to see the two files 'fred' and 'joe' on the terminal. We could say:

    cat <fred

and then:

    cat <joe

This is time consuming and not at all appropriate. We can't say:

    cat <fred <joe

because there is only one standard input.

Many tools, therefore, accept files names as parameters, when this cannot lead to any ambiguity. The tool 'cat' is one such tool. The name 'cat' comes from an abbreviation of 'cat's function: 'concatenate' — the original purpose of the 'cat' tool was to concatenate files. The files that are to be concatenated are specified as parameters to the 'cat' tool. This does not affect the ideas of standard input and standard output: it augments them. So it is possible to say:
cat fred joe

which means "put the file 'fred' on to standard output, and then put the file 'joe' after it." Notice that standard output was not redirected and so the terminal will be used. What happen to standard input? It would be very inconvenient to have to use the terminal as a third input file, so tools that accept file names as input parameters will not accept input from standard input if a file name is specified.

Note: Skip on first reading

To specify standard input when a file name is present the dash, '-' is used. In this way, the command:

cat fred - joe

means:

(1) put file 'fred' on the standard output.

(2) put whatever is typed in on the terminal (standard input) on to standard output.

(3) put the file 'joe' onto standard output.

The command:

cat fred - joe >harry

will do the same as above, except that it will redirect the standard output file to the file 'harry'.

3. Error output

If when executing a tool, you encounter an error, the error output goes by default to the terminal. In order to redirect the error output into a file, you can do the following:

cat fred ? joe

This means that if an error occurs while the 'cat' tool is getting input from the file 'fred', the error message goes to the file 'joe' instead of the terminal.

4. Pipes

The chief value of having standard input and output is the ability to connect the output of one program so as to be the input to the next. This allows the user to connect together the very simple programs that make up the Software Tools into very complex and useful programs -- all by only using the command language. In an earlier example we redirected the output of the
'date' command to a file 'fred', and then changed all ':'s to '/'. To avoid the temporary file 'fred' we can pipe the output of 'date' directly to 'ch'.

The notion of the 'pipe' here is analagous to the idea of a pipe in plumbing. We use the pipe to connect the output of one tool to be the input of the next. The input and output files are, of course, standard input and standard output.

The pipe is represented by the vertical bar '|'. So the command that prints the date with slashes as separators (instead of the colon) will be:

```
    date | ch ':' '/'
```

To redirect the final output to the file 'joe' as we did earlier, we say:

```
    date | ch ':' '/' >joe
```

Pipes can be put together into long chains of commands. Obviously, commands whose output is input to a pipe cannot redirect their output elsewhere, and commands whose input is to come from a pipe cannot independently redirect their standard input.

5. Appending To Files

Often, we want to append some data at the end of a file instead of completely replacing it. To do this we use a pair of greater than signs, '>>', with no space between them. To put the output from the 'date' command on to the file 'fred' we say:

```
    date >>fred
```

To append the date to the file 'fred' we say:

```
    date >>fred
```

Appending the output of a tool to a file may be used whenever standard output may be redirected. Obviously, output cannot be redirected independently.

A series of commands can be used together to create a file. For example, if we want to put the date at the beginning and the end of a file 'mary' and put this result on the file 'connie'. We could do this by:

```
    date >connie; cat mary >>connie; date >>connie
```

Since it is fairly tedious to respecify the file connie over and over, the shell allows us to put these commands together by placing them in parentheses, thus:

```
    (date; cat mary; date) >connie
```
The shell will generate the appends to the file 'connie' automatically. (This is discussed in more detail in the shell tutorial).

Error output can also be appended. For example if you did the following:

```
cat fred ? joe
```

and an error occurred. The file 'joe' now contains the error output. In order to append the file 'joe', with other error output, the following must be done:

```
cat tom ?? joe
```

If an error occurred in the reading of the file 'tom', the error output would be appended to the file 'joe'.

6. Here Documents

Often we want to include some text as a part of a command. For large amounts of text, we can put the text on a file and use an appropriate tool to put it in the right place. For very small amounts of text, we can use the 'echo' tool. For medium volumes, say a few lines, we can use a 'here' document. A 'here' document is a way of providing small amounts of constant data to programs. In our previous example, we wanted to add the date to the beginning and the end of a file 'mary' and put the output on the file 'connie'. If the file 'mary' is small, and it is not going to be reused, we need not define the file 'mary' but use a 'here' document instead.

A 'here' document is defined by using two less than characters, '<<', followed by a delimiter character of your choosing. (If special shell meta-characters are used, they must be escaped with the at '@' sign Some shell meta-characters are '?', '<', '>', '|', and '$'.) Often an exclamation mark, '!', is used. If your file 'mary' contained the following three lines:

```
a to z
A to Z
0 to 9
```

the results of the previous example could be achieved by

```
date >connie
cat <<! >>connie
a to z
A to Z
0 to 9
!
date >>connie
```

or:

```
(date; cat <<!; date) >connie
a to z
A to Z
```

-5-
7. Summary

Redirection is a central idea in the Software Tools: it allows tools to be written that can be combined together with other tools to make an extremely powerful environment for program development and maintenance. Every tool that has input or output has a standard input and output defined. The default is the terminal in both cases. Input and output can be redirected in one of the following ways:

- > put the output of this tool onto the file specified
- < read the input for this tool from the file specified
- >> append the output of this tool onto the file specified
- << read the input for this tool from the command stream
- | take the output of the tool before this symbol and use it as the input to the tool following this symbol
- - use standard input here, along with files are specified as parameters

FILES
none

SEE ALSO
none
NAME
regexp - explains how to use regular expressions

SYNOPSIS
(see below)

DESCRIPTION

INTRODUCTION

Regular expressions are a fundamental concept in the Software Tools. A regular expression is simply a pattern which is used to match strings. Regular expressions are used by several tools, the chief of which are 'ch' (for 'change'), find, and ed (the editor). The concepts of regular expressions are also used in 'tr' (for 'transliterate'). For details on the operations of these tools, see the respective manual entries.

1. Simple Regular Expressions

The simplest regular expression is simply an alphanumeric character string, such as:

'abc'

To find all files in current directory that have the string 'abc' as a a part of their name, say:

ls | find 'abc'

The 'ls' command will list all the files in the current directory, and the tool 'find' will put onto standard output all lines that have an occurrence of the regular expressions given as arguments. The above command will print, on the terminal, all file names that have the string 'abc' anywhere in their name.

To find all the files that have 'a', 'b', or 'c' in them, the command would be:

ls | find 'a' 'b' 'c'

This time the 'find' command has been passed three simple regular expressions as arguments, and its job is to find all lines that have occurrences of any of these three regular expressions in them. Any characters may be specified in these simple regular expressions except for a few characters that have special meanings. These characters are called metacharacters. The rest of this document is concerned with the uses and meanings of the metacharacters. A complete list of all metacharacters is given in the last section of this document.
2. Simple Metacharacters

Let's say that we're interested in files that start with the characters 'abc', rather than just have 'abc' anywhere in the name. To do this we use the metacharacter '%', which means the *beginning* of the line. For example:

```
ls | find '%abc'
```

will find all lines that start with 'abc'.

The companion of the percentage sign is, somewhat appropriately, the dollar sign. The dollar sign, '$', signifies the *end* of a line, so to find all files that end with 'abc', we say:

```
ls | find 'abc$
```

Metacharacters can be mixed freely, so to find the file 'abc' with nothing else on the line, we say,

```
ls | find '%abs$
```

The next useful metacharacter is '?' which means match any single character, except the end of line which is the metacharacter '$'. For example, to find all single character file names, we would say:

```
ls | find '%?$
```

Note that:

```
ls | find '?$
```

isn't going to match just single character file names, because all file names that have at least one character will match the regular expression '?', but we could make a construction of this form, such as:

```
ls | find '??????'
```

to find all file names that have at least six characters in them.

The next important metacharacter is '*' which is used to indicate repetition. The asterisk is used to mean zero or more repetitions of the last regular expression component. (We'll define a regular expression component later). For example:

```
ls | find '%a*$
```

will find files whose names consist solely of a's. Note, however, that this regular expression will also match a blank line, because the repetition character covers zero or more occurrences of the character. So:

```
ls | find '%abcx*$
```
will find the files

```
abc
abcx
abcxx
abcxxx
```

and so on.

One common use of the repetition metacharacter is in conjunction with the 'match any character' metacharacter, as in '?*'. This is useful when you don't care what comes between two important strings. For example, to find all files that start with 'ab' and end in 'cd' we say:

```
ls | find '%ab?*cd$'
```

Note that this will find the line 'abcd', because the repetition metacharacter includes zero repetitions.

3. Problems - Part 1

Let's say we have a file 'fred' which contains:

```
a
ab
xyz
axyz
abxyzcd
abxyabcd
xyzcdx
abxyzcd
```

What will the following commands yield?

1. `find <fred 'z'
2. `find <fred '%x'
3. `find <fred 'z$'
4. `find <fred '%ab$'
5. `find <fred '%ab?$'
6. `find <fred '%ab?*
7. `find <fred '%?*xyz?*
8. `find <fred '%?*xyz??'
9. `find <fred '?*xyz?*d'

-3-
(10) find <fred 'ab?*cd'

How do you write regular expressions to get:

(1) all lines that contain 'ab'?

(2) all lines that contain an 'x' at the end of a line?

(3) all lines that start with 'ab' and end in 'cd'?

(4) all two character lines?

(5) all lines with at least six characters?

(6) all lines that contain 'xy' and have at least one character before and after it?

4. Escaping Metacharacters

What if we want to find an occurrence of one of the metacharacters, such as the dollar sign? We can't say:

    find '$'

because this will find all lines that have an end to them! We need a way to tell the system that we mean '$' and not its special meaning -- end of line. We need a way to escape the special meaning of the character. This is done by preceeding the character with an 'at sign', '@'. So to find occurrences of the dollar sign, we say:

    find '@$'

The 'at sign' is another metacharacter whose meaning is 'reverse the meaning of the next character'. It does not mean 'interpret the next character as being a simple character and not a metacharacter'. This convention is used throughout the Tools -- it is even used on itself! Let's say we want to look for an occurrence of the at sign. We say:

    find '@@'

where the first '@' means 'reverse the meaning of the next character', and so the second '@' is interpreted as the character '@' itself.

Some of the alphabetic characters can also be escaped to specify certain non-printing characters. These are:

    @n - newline
    @b - backspace
    @d - delete
    @t - tab
    @f - form feed
5. Character Classes, Ranges and Negation

We saw at the beginning that 'find' will accept multiple arguments and try to find the strings that match any of the regular expressions given as arguments. Obviously, in the editor, we can't say:

/a b c

because that means "find a <space> b <space>" and not "find an occurrence 'a' or 'b' or 'c'". The way we solve this is by the notion of a 'character class', which is simply a way of saying 'any one of these'.

A character class is denoted by square brackets ('[ ' and ' ]') surrounding the characters in question. For example:

find '[abc]'

will find all occurrences of 'a' or 'b' or 'c' in the same way as:

find 'a' 'b' 'c'

So the editor command:

/[abc]

will find an occurrence of 'a' or 'b' or 'c'.

The command:

ls | find '[pq]xyz'

will find files that have occurrences of 'pxyz' or 'qxyz' in them. Characters classes can be put next to each other, as in the editor command:

/[Gg][Ee][Tt]

which will address the line that contains any of the eight possible combinations of upper and lower case occurrences of 'GET'.

What if we want to express the idea of finding one of a large number of things, such as an upper case character? We could enumerate them, but that is tedious and error-prone. Character classes provide the ability to specify ranges of characters. A range is specified by putting a dash between the two characters at the extremes of the range. (The collation sequence is ASCII). For example:

[0-9]

is the same as

[0123456789]
[A-Z]

means 'any uppercase character'. Several ranges can be specified in a character class, so to find any letters (upper or lower case) we can say:

find '[A-Za-z]'

Note that there is no separator between the ranges.

What if we want to find an expression that does not contain a number. The negation character, '! ' is used to specify a negated character class A negated character class specifies that any character will match except those specified in the character class. For example, to find all files that do not contain digits, we can say:

ls | find [!0-9]

6. Metacharacters in 'ch'

The tool 'ch' is used to change patterns to strings, one line at a time. For example,

ch <fred 'abc' 'xyz' >joe

will change every occurrence of 'abc' to 'xyz' everywhere in the file fred, and put the result on the file 'joe'. The first parameter, in this example, is a regular expression. Of course, the previous examples will also work in 'ch'. The command:

ch 'ab?*cd' 'xyz'

will replace every occurrence of 'ab' followed by anything and then 'cd' by 'xyz'.

The 'ch' tool also recognizes some additional metacharacters. The 'ch' tool recognizes the metacharacter '&' which means 'take the regular expression previously found and use it'. For example, to replace each occurrence of 'abc' by 'xyzabcxyz' we could say:

ch 'abc' 'xyzabcxyz'

or

ch 'abc' 'xyz&xyz'

In this case, this is merely a shorthand convenience. Similarly, if we want to put xyz at the beginning and end of every line, we could say:

ch '%' 'xyz' | ch '$' 'xyz'

but we can also do this with a single command:
Because "?*" will always match the longest possible pattern, we could also have said:

ch '?*' 'xyz&xyz'

This use of the ampersand is very useful when several operations need to be performed on a list of files. If we want to print all files in the current directory on the line printer we can say:

ls | ch '?*' 'pr & | lpr' | sh

which is a type of command we see often.

This ability to use the last pattern found is so useful that it soon becomes valuable to allow several patterns to be used. The generalization is to surround regular expressions by escaped parentheses, and then address each sub-regular expression by an escaped number. For example, the following is equivalent to the previous example above:

ls | ch '@(?*@)' 'pr @0 | lpr' | sh

Each following regular expression in parentheses is addressed by @0, @1, @2, @3, and so on. To swap two patterns separated by spaces we can say:

ch '@(?*@) @(?*@)' '@1 @0'

7. Answers to Problems

7.1. Problems - 1

The answers to the problems in Section 4, Part 1 follow:

(1) The answer to problem (1) is:

   xyz
   axyz
   abxyzcd
   xyzcdx
   abxyzcd

(2) The answer to problem (2) is:

   xyz
   xyzcdx
(3) The answer to problem (3) is:

    xyz
    axyz

(4) The answer to problem (4) is:

    ab

(5) The answer to problem (5) is:

    [no lines match]

(6) The answer to problem (6) is:

    ab
    abxyzcd
    abxyabcd
    abxyzcd

(7) The answer to problem (7) is:

    xyz
    axyz
    abxyzcd
    xyzcdx
    abxyzcd

(8) The answer to problem (8) is:

    abxyzcd
    xyzcdx
    abxyzcd

(9) The answer to problem (9) is:

    abxyzcd
    xyzcdx
    abxyzcd

(10) The answer to problem (10) is:

    abxyzcd
    abxyabcd
    abxyzcd
7.2. Problems - 2

The answers to the problems in Section 4, Part 2 follow:

(1) find 'ab'
(2) find 'x$'
(3) find '%ab?*cd$'
(4) find '%??$'
(5) find '??????'
(6) find '?xy?'

SEE ALSO

DIAGNOSTICS
NONE

AUTHOR
S.J. Mellor, with modifications by Ricky Barnes

BUGS/DEFICIENCIES
NAME
sh - shell (command line interpreter)

SYNOPSIS
sh [-xvna] [script [arg1 ... [arg9]]]
or sh [-xvna] -c argline

DESCRIPTION

1. Purpose of the Shell

Sh (also called 'the shell') executes commands, which may in turn cause execution of other commands.

2. Invoking the shell

'Script' is the name of the file to read commands from. The shell uses the +PATH search path environment variable to locate 'script' if necessary. 'Arg1' through 'argn' are the arguments to be given to the command file 'script'. If the 'script' argument is omitted, sh reads the commands from standard input.

3. Shell Flags

The following flags are meaningful to the shell:

-x echo lines as the shell reads them

-v report what the shell is doing as it is doing it

-n process the commands given, but don't actually execute them

-a the shell should continue executing commands, and will not abort, even if it is unable to spawn a tool, or it spawns a tool and that tool aborts.

-c the next argument is a command line: the shell will only execute this command line, and will not read commands from standard input

4. Typing in a Command Line

When the shell is ready to execute a command from the terminal, it prompts with a '%' . You can then type a command, terminating it with a carriage return. If you cannot finish typing the command on one line, you can type a single atsign and a carriage return, and the shell will prompt you to continue typing the rest of the command.
5. Commands

The simplest possible command that a user can give to the shell is a program name followed by optional arguments. If you give this type of argument, the shell will execute the program with the name specified. The program name and the arguments are all separated from each other by spaces. For example, you might type

```sh
echo Hi There!
```

This tells the shell to execute the program called 'echo', with the two arguments 'Hi' and 'There!'. The echo tool simply 'echoes' its arguments back to you. The output from the command 'echo Hi There!', for example, would be

Hi There!

The name 'echo' actually is the name of the file where the echo program can be found. Typically, the location of the echo program is in a file like '/usr/bin/echo'.

When the 'echo' program finishes executing (it shouldn't take very long), the shell will type the prompt sign, %, to indicate that it is ready to execute another command.

6. Search Paths

You may have wondered how the shell knew that the 'echo' program was in the '/usr/bin' directory in this example. You can give the shell a list of directories in which to search for programs. This list is specified by setting the environment variable 'PATH'. Several programs besides the shell use search paths. For example, the programs 'rat4', 'ld', and 'man' use (other) search path variables. Type 'man paths' or 'man environ' for more information.

If you want to be completely specific about which copy of the program 'echo' you runs, you could can specify the entire file name of the program: for example, you could type:

```sh
/usr/bin/echo Hi There!
```

7. Arguments

Arguments may be given as sequences of characters, as strings quoted with double or single quotes, as parameter substitutions, as escape sequences, or as any combination of these.

The simplest way of specifying an argument is to simply take the characters of the argument exactly as you would like the program to see it. (For example, 'Hi' and 'There!' as given above.)
This method of passing arguments has certain disadvantages, however. The main problem is that if a character that is special to the shell (such as `;` `<` `>` `?` or space) is given in an argument, the shell is likely to misinterpret it. To specify that you want the shell to interpret a character as part of an argument rather than one of the shell's special characters, simply precede the special character by an atsign. For example, the character sequence @; would put a semicolon in an argument, and @@ would put an at sign in it. In addition, if the escape character @ is followed by any of the following characters, the string @x is replaced by another hard to specify character:

- @b = backspace
- @s = space
- @t = tab
- @f = formfeed
- @d = delete

For any other character x, @x is replaced by x.

Also, if you want to pass a whole series of characters that might be misunderstood by the shell, you may enclose them in single or double quotes. For example, if you wanted to pass `roft` a subtitle, say

```
Act III - The Denouncement
```

the shell would think that the spaces separating the words of the argument were actually supposed to be separating separate arguments: 'Act', 'III', '-', and so on. If you wrote, instead

```
"Act III - The Denouncement"
```

the shell would understand that the entire title is a single argument. It usually can't hurt to enclose a string in quotes if you aren't sure if the shell will interpret it correctly. $n parameter substitutions (see below) are performed in double quoted strings, but not in single quoted ones. Other than that, any character in a string except the character that it is quoted with will be passed intact to the program you are executing.

Arguments to "script files" (see below) are also allowed. In a script, a sequence $n (where n > 0) is replaced by the n'th argument that the script was invoked with. (Remember, however, that on the command line, $n parameter substitutions are not performed inside single quoted strings.)

8. Comments

Any command which begins with a '#' character (possibly preceded by spaces) is a comment and is ignored by the shell. Blank lines are also ignored.
9. The Goto Statement

Inside a script, you can tell the shell to go to another part of the script by using the `goto` command. The `goto` command takes one argument. For example:

```
goto label
```

This command would tell the shell to go to the line in the script with the label 'label'.

10. Labels

Labels are denoted by lines which begin with a colon, ':', followed by one or more spaces, then the label name. Label lines are ignored except when the shell executes `goto`s. Labels may be made of any combination of letters, digits, and underlines. For example:

```
echo starting script
: repeat_forever
  echo Hi There!
  goto repeat_forever
```

will echo its message 'Hi There!' forever.

11. Redirection

Most tools have a primary input and a primary output. Unless you specify something to the contrary, these tools programs have their primary input from the terminal, and their primary output to the terminal.

For example, if you say

```
find 'ance$' 'ence$'
```

the find tool will read lines from the terminal. And whenever the find tool reads a line which ends with 'ance' or 'ence', it types this line on the terminal.

You might think that this isn't very useful. That's true: there is little point for find to tell you which of the lines you typed end in 'ance$' or 'ence$'. However, suppose you frequently misspelled words ending in 'ance' or 'ence', and you wanted to see all these words. So if you could only read from the dictionary file in /etc/lexicon, you could find these words. To do this, type

```
find 'ance$' 'ence$' < /etc/lexicon
```

Now, suppose that since you need to refer to this list often, you want to put the list in the file 'spelling'. You could type:
The primary input file is called 'standard input', and the primary output file is called 'standard output'. There is also a third file in which tools programs write many of their error messages: this file is called 'error output'. The practice of having 'standard input', 'standard output' or 'error output' go to some place other than their default is called 'redirection'.

Generally, given a command, you can have that command read from some file 'infile' by typing

```
command < infile
```

Also, you can have the command write to some file 'outfile' by typing

```
command > outfile
```

Similarly, you can have the command write its error messages to some file 'errfile' by typing

```
command > errfile
```

Of course, you can use any combination of these as well. If the files 'outfile' or 'errfile' don't exist they are created.

12. Append Access

Occasionally, it is useful to append to a file (add to the end of the file) rather than write to it. To append the tool's standard output to the file 'outfile', you can type

```
command1 > > outfile
```

Similarly, to append the error messages to the file 'errfile', you can type

```
command1 > errfile
```

Like normal redirection, if 'outfile' or 'errfile' don't exist, they are created.

13. Pipes and Filters

Often, you may want to have the output of one command become the input of another. You could always type

```
command1 > scrat
command2 < scrat
rm scrat
```

(where 'scrat' is some scratch file). However, there is an easier way:

```
command1 | command2
```
Offhand, this doesn't sound very useful. But the more you use the software tools, the more useful you will realize it is. Suppose you have a file made up of a list of words, and you want to see which of these words are not in the dictionary. To do this type

```
sort words | uniq | comm -2 /etc/lexicon
```

Here, the 'sort' program sorts the file 'words', then the 'uniq' program removes all the duplicate words, and then the 'comm' program prints all those words that are in the list with the duplicates removed which are not in the dictionary /etc/lexicon. As a matter of fact, this is basically the way the 'spell' tool (actually a script) works.

Programs such as 'find', 'ch', 'sort', 'uniq', and 'comm' which read from standard input and write to standard output performing transformations on the input are called 'filters'.

14. Compound Commands

You can make two commands 'command1' and 'command2' into a single command

```
command1 ; command2
```

For example, if you wanted to copy the file '/usr/src/rat4' to your space and then edit it, you could type

```
cp /usr/src/rat4 src/myrat4 ; ed src/myrat4
```

15. Parenthesis

You might ask 'why not just type the 'cp' command and then type the 'ed' command?' In this case there is no reason why typing ';' would be better than typing separate commands. What really makes ';' useful however, are parenthesis.

Everything inside parenthesis is executed as a unit. For example, say you had two lists 'list1' and 'list2', and you wanted to combine the things which end with 'abc' in list1 with the things which end end with 'xyz' in list2 and sort them. A command which would do this is

```
( find 'abc$' < list1 ; find 'xyz$' < list2 ) | sort
```

16. Execution in the Background

On some systems, the shell will execute processed in the 'background', so that you can do something else while this one process is running. To execute
a command 'command' in the background, type
   
   command &

Say you are compiling the shell, and you wanted to do something else while it
is compiling. Say

   rc src/cat &

The shell will return immediately, and will print a number called the 'process
number'. This process number can be used later to kill the process if you
don't want it to run anymore.

17. Scripts

If you have a collection of commands that you execute frequently, you can
put them in a file. Such a file made up of commands to the shell is called a
'shell script' or simply a 'script'.

As mentioned before, any occurrences of $n in the commands in the script,
where the $n is outside of single quotes, will be replaced by the n'th argu-
ment.

For example, suppose you frequently need to execute the following com-
mands:

   tr A-Z a-z < [some file name] | tr !a-z '@n' | @
   sort | uniq | comm -2 /etc/lexicon

(This script would take a file, convert it to lower case, split words onto
separate lines, ignoring anything except letters, sort it, so that we have a list
of all the words in the file. It would then remove all the duplicate words, and
would then see which of these words are not in the dictionary. The result of
all this is to catch spelling errors.)

You could put the commands

   tr A-Z a-z < $1 | tr !a-z '@n' | @
   sort | uniq | comm -2 /etc/lexicon

in a file called 'myspell' in your directory, and whenever you wanted to see
all the spelling errors in some file (say 'text') you could say

   myspell text

(You don't actually have to use this script because there is a already a spell
tool implemented. Say 'man spell' for details.)
18. Here documents

Inside a script, you may want to specify particular data to be input to a command. The way you can do this is by using a 'here document':

```bash
command << !
data...
!
```

This will cause the data 'data ...' to be the information in the standard input for the command 'command'. The character '!' may be any printing ascii character. (It may have to be escaped on the command line, however.) At signs and $n parameters are evaluated inside of here documents.

For example, if you wanted to write a script which would use the editor to edit a file, you could write a script like:

```bash
ed - $1 <<!
   /section1/,/section2/gs/ebcdic/ascii/g
   wq
!
```

Actually, it is possible to have multiple here documents in a command. When this is done, the data for the here documents must follow the entire command, and here documents are associated with the command as the command is read from left to right. An example will make this clearer:

```bash
command1 << ! ; command2 << =
data for command1...
!
data for command2...
```

19. Reserved Shell Commands

The shell executes certain commands directly. The shell doesn't search for these commands, so if the user has a command with the same name as one of these commands, it may not be found. See the corresponding manual entries (tool section) for information on these commands.

```bash
cd change working directory
if execute a command, if some condition is true
goto resume execution of a script at the point specified by a label. (See section Goto above for more information.)
```
exit   terminate the execution of a shell script

logout exit the shell. The user can exit the shell by typing an end of file, as well as by typing 'logout'.

There are also built-in commands to manipulate environment variables. See the manuals for setenv, resetenv, and unsetenv.

20. Abort Behavior

The shell will always abort if the script specified on the command line 'sh [flags] script' was not found, if it isn't ascii, if internal limits are reached, or if some other unexpected thing (such as not being able to create a temporary file) happens.

Normally, the shell will also abort if it is reading its commands from something other than a terminal, and if a tool that the user told the shell to execute aborts or can't be found. In this case, the shell does not execute any more commands, even if they appear on the command line on which the tool aborted or couldn't be found.

If the user specifies the '-a' flag, the shell will continue executing commands, and will not abort, even if something can't be found or something aborts. Instead, it will continue to execute as if the tool was found, and exited without aborting. Remember, however, that files that were expected to be created may not have been created, and so on.

21. User Requested Aborting

On most machines, the user can decide to abort a tool while it's executing. For example on the vax, you can abort a tool with "-C", and on the modcomp, you can abort a tool with "^Aa<NEWLINE>". This sequence ("^C" or "^Aa<NEWLINE>") is called the abort sequence for the given computer. The abort sequence always aborts the tool or script in the foreground which was most recently executed.

For example, if you were running a script 'argh' which ran the tool 'foobar' (which hasn't spawned anything), and typed the abort sequence, the tool 'foobar' would abort. It would then be up to the script 'argh' to decide what to do.

On the other hand, if the script argh wasn't running a tool at the moment you type the abort sequence, the script 'argh' would abort. In that case, it would be up to whatever tool or script which executed the script 'argh' to decide what to do.

(To be more precise, the most recently spawned foreground process is aborted, where a process is said to be a foreground process if it is your login process, or if it was spawned by a foreground process in WAIT mode. (Any
other process is called a *non-foreground process.*

**SEE ALSO**

files(info), paths(info), environ(file) cd(tool), if(tool), goto(tool), exit(tool),
logout(tool) sh(tool), setenv(tool), resetenv(tool), unsetenv(tool)

**DIAGNOSTICS**

<tool>: not found

the program or script <tool> could not be found. <tool> will be a full path name, either or the form /x/y/z... or +PATH/<name>. (+PATH contains the search path the shell tried to use to find the tool.)

Various other error messages are printed when unsyntactical commands are given to the shell.

**AUTHOR**

Originally written be Dennis Hall, Joe Sventek, and Debbie Scherrer. Modified by Bob Upshaw, Vern Paxson, Todd Hammond and others. Rewritten using the Software Tools yacc and lex by Marshall Spight.
NAME

YACCTUT - how to construct a yacc input specification

SYNOPSIS

see below

DESCRIPTION

0. Introduction

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the "lexical analyzer") to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called "grammar rules". When one of these rules has been recognized, the user code supplied for this rule, called an action, is invoked; actions have the ability to return values and make use of the values of other actions.

Yacc is written in RATFOR and the actions, and output subroutine, are in RATFOR as well. Moreover, many of the syntactic conventions of Yacc follow RATFOR.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

date : month_name day ',' year ;

Here, date, month_name, day, and year represent structures of interest in the input process; presumably, month_name, day, and year are defined elsewhere. The comma "," is enclosed in single quotes; this implies that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule, and have no significance in controlling the input. Thus, with proper definitions, the input

July 4, 1776

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user-supplied routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a "terminal symbol", while the structure recognized by the parser is called a
"nonterminal symbol". To avoid confusion, terminal symbols will usually be referred to as tokens.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the rules

```plaintext
month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;
...
month_name : 'D' 'e' 'c' ;
```

might be used in the above example. The lexical analyzer would only need to recognize individual letters, and month_name would be a nonterminal symbol. Such low-level rules tend to waste time and space, and may complicate the specification beyond Yacc's ability to deal with it. Usually, the lexical analyzer would recognize the month names, and return an indication that a month_name was seen; in this case, month_name would be a token.

Literal characters such as "","" must also be passed through the lexical analyzer, and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule

```plaintext
date : month '/' day '/' year ;
```

allowing

```
7 / 4 / 1776
```

as a synonym for

```
July 4, 1776
```

In most cases, this new rule could be "slipped in" to a working system with minimal effort, and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan; thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found. Error handling, provided as part of the input specifications, permits the reentry of bad data, or the continuation of the input process after skipping over the bad data.

In some cases, Yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self-contradictory, or they may require a more powerful recognition mechanism than that available to Yacc. The former cases represent design errors; the latter cases can
often be corrected by making the lexical analyzer more powerful, or by rewriting some of the grammar rules. While Yacc cannot handle all possible specifications, its power compares favorably with similar systems; moreover, the constructions which are difficult for Yacc to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid Yacc specifications for their input revealed errors of conception or design early in the program development. The theory underlying Yacc has been described elsewhere [1,2,3].

The next several sections describe the basic process of preparing a Yacc specification; Section 1 describes the preparation of grammar rules, Section 2 the preparation of the user supplied actions associated with these rules, and Section 3 the preparation of lexical analyzers. Section 4 describes the operation of the parser. Section 5 discusses various reasons why Yacc may be unable to produce a parser from a specification, and what to do about it. Section 6 describes a simple mechanism for handling operator precedences in arithmetic expressions. Section 7 discusses error detection and recovery. Section 8 discusses the operating environment and special features of the parsers Yacc produces. Section 9 gives some suggestions which should improve the style and efficiency of the specifications. Section 10 discusses some advanced topics, and Section 11 gives references. Appendix A has a brief example, and Appendix B gives an example using some of the more advanced features of Yacc.

1. Basic Specification

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. In addition, for reasons discussed in Section 3, it is often desirable to include the lexical analyzer as part of the specification file; it may be useful to include other programs as well. Thus, every specification file consists of three sections: the declarations, "(grammar) rules", and programs. The sections are separated by double percent "%%" marks. (The percent "%" is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is
Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments begin with a hash mark, '#', as in RATFOR.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

\[ A : \text{BODY} ; \]

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation.

Names may be of arbitrary length, and may be made up of letters, dot '.', underscore '_', and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes '. '. The at sign '@' is an escape character within literals, and the following escapes are recognized:

- '@n'  newline
- '@r'  return
- '@'  single quote '. '
- '@@'  at sign '@'
- '@t'  tab
- '@b'  backspace
- '@f'  form feed
- '@C'  C, where C is any other character

If there are several grammar rules with the same left hand side, the vertical bar '|' must be used instead of rewriting the left hand side. In addition, the semicolon at the end of a rule is dropped before a vertical bar. Thus the grammar rules

\[
A : B C D ; \\
A : E F ; \\
A : G ;
\]

would be given to Yacc as

\[
A : B C D \\
| E F \\
| G \\
|
\]

It is necessary that all grammar rules with the same left side appear
together in the grammar rules section.

If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:

```plaintext
empty :
```

Names representing tokens must be declared; this is most simply done by writing

```plaintext
%token name1 name2 ...
```

in the declarations section. (See Sections 3, 5, and 6 for much more discussion). Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, one, called the "start symbol", has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section.

The end of the input to the parser is signaled by a special token, called the endmarker. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the endmarker is seen; it accepts the input. If the endmarker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

2. Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is an arbitrary RA.TFOR statement, and as such can do input and output, call subroutines and functions, and alter external variables. An action is specified by one or more statements, enclosed in "%{" and "%}". For example,

```plaintext
A : '(' B ')' %{
    call prompt( pstring, STDOUT )
}
```
and

```
XXX : YYY ZZZ
%
   call putlin( message, ERROUT )
   flag = 25
%
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol "dollar sign" "$" is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable "$\$" to some value. For example, an action that does nothing but return the value 1 is

```
%
$$ = 1
%
```

The right side of each rule contains terminal and non-terminal symbols. Each of these symbols has a value associated with it. If the symbol is a terminal symbol, then the value associated with it is the value that the lexical analyzer returned when it recognized this symbol. A non-terminal symbol has the value that was returned by the action associated with the rule which defines this non-terminal symbol. To obtain the values associated with each right side symbol, the action may use the pseudo-variables $1, \$2, \ldots$, which refer to each of the right side symbols, reading the rule from left to right. Thus, if the rule is

```
A : B C D
```

for example, then \$2 is the value associated with C, and \$3 the value associated with D.

As a more concrete example, consider the rule

```
expr : '(' expr ')'
```

The value returned by this rule is usually the value of the expr in parentheses. This can be indicated by

```
expr : '(' expr ')'
%
$$ = $2
%
```
By default, the value of a rule is the value of the first element in it ($1). Thus, grammar rules of the form

$$A : B$$

frequently need not have an explicit action.

In many applications, output is not done directly by the actions; rather, a data structure, such as a parse tree, is constructed in memory, and transformations are applied to it before output is generated. Parse trees are particularly easy to construct, given routines to build and maintain the tree structure desired. For example, suppose there is a RATFOR function `node`, written so that the call

$$\text{index} = \text{node}(L, n1, n2)$$

creates a node with label $L$, and descendants $n1$ and $n2$, and returns the index of the newly created node. Then parse tree can be built by supplying actions such as:

```yacc
expr : expr ' + ' expr
    %{
    $$ = \text{node}( '+', $1, $3 )
    %}
    ;
```

in the specification.

The user may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section, enclosed in the marks "%~" and "%1". These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example,

```yacc
%{
integer variable
data variable /0/
%
```

could be placed in the declarations section, making `variable` accessible to all of the actions. The Yacc parser uses only names beginning in "yy"; the user should avoid such names.

In these examples, all the dollar values are integers, values of other types must be accessed through integer pointers.

3. Lexical Analysis
The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function with one argument. The function returns an integer, the "token number", representing the kind of token read. If there is a value associated with that token, it should be assigned to yylex's single argument. The synopsis for yylex is:

\[
yytoken = \text{yylex}( \text{yyvalue} )
\]
\[
\text{yytoken - next token on input stream}
\]
\[
\text{yyvalue - value of yytoken}
\]

The parser and the lexical analyzer must agree on the token numbers in order for communication between them to take place. The token numbers may be chosen by Yacc, or chosen by the user. In either case, the "define" mechanism of RATFOR can be used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the Yacc specification file. The relevant portion of the lexical analyzer might look like:

```fortran
integer function yylex( value)
    integer value
    integer i, atoi
    character tokstr( MAXLINE ) #next token string
    call gettok( tokstr ) #get next token string
    switch ( tokstr(1) )
    {
      . . .
      case DIG0, DIG1, DIG2, DIG3, DIG4,
          DIG5, DIG6, DIG7, DIG8, DIG9:
        {
          i = 1
          value = atoi( tokstr, i )
          return DIGIT
        }
      . . .
    }
```

The intent is to return a token number of DIGIT, and a value equal to the numerical value of the digit. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT. This mechanism leads to clear, easily modified lexical analyzers. The token name error is reserved for error handling, and should not be used naively (see Section 7).

As mentioned above, the token numbers may be chosen by Yacc or by the user. In the default situation, the numbers are chosen by Yacc. The default
token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 259.

To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately followed by a positive integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition. It is important that all token numbers be distinct.

For historical reasons, the endmarker must have token number equal to 0. The user can use the RATFOR definition YYENDTOK. It has been pre-defined to be zero. This token number cannot be redefined by the user; thus, all lexical analyzers should be prepared to return 0 (or YYENDTOK) as a token number upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the Lex tool. This lexical analyzer is designed to work in close harmony with Yacc. The specifications for Lex use regular expressions instead of grammar rules. Lex can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as FORTRAN) which do not fit any theoretical framework, and whose lexical analyzers must be crafted by hand.

4. How the Parser Works

Yacc turns the specification file into a RATFOR program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex, and will not be discussed here (see the references for more information). The parser itself, however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The "current state" is always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 1, the stack contains only state 1.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls the routine yylex to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser
decides on its next action, and carries it out. This may result in states
being pushed onto the stack, or popped off of the stack, and in the loo­
kahead token being processed or left alone.

The *shift* action (also called a *transition*) is the most common action the
parser takes. Whenever a shift action is taken, there is always a lookahead
token. For example, in state 56 there may be a shift action to state 34, and
in state 34 the token just seen is IF. This means that, in state 56, if the looka­
head token is IF, the current state (56) is pushed down on the stack, and
state 34 becomes the current state (on the top of the stack). The lookahead
token is cleared.

The *reduce* action keeps the stack from growing without bounds. Reduce
actions are appropriate when the parser has seen the right hand side of a
grammar rule, and is prepared to announce that it has seen an instance of
the rule, replacing the right hand side by the left hand side. It may be
necessary to consult the lookahead token to decide whether to reduce, but
usually it is not.

Suppose the rule being reduced is

\[ A : x \ y \ z : \]

The reduce action depends on the left hand symbol (A in this case), and the
number of symbols on the right hand side (three in this case). To reduce,
first pop off the top three states from the stack (The number of states
popped equals the number of symbols on the right side of the rule). In
effect, these states were the ones put on the stack while recognizing \( x \), \( y \), and
\( z \), and no longer serve any useful purpose. After popping these states, a
state is uncovered which was the state the parser was in before beginning to
process the rule. Using this uncovered state, and the symbol on the left side
of the rule, perform what is in effect a shift of \( A \). A new state is obtained,
pushed onto the stack, and parsing continues. There are significant
differences between the processing of the left hand symbol and an ordinary
shift of a token, however, so this action is called a *goto* action. In particular,
the lookahead token is cleared by a shift, and is not affected by a goto. In
any case, the uncovered state contains a transition state, which is the state
that has just recognized \( A \) (i.e. *shifted* \( A \)). This transition state is then
pushed onto the stack, becoming the current state.

In effect, the reduce action "turns back the clock" in the parse, popping the
states off the stack to go back to the state where the right hand side of the
rule was first seen. The parser then behaves as if it had seen the left side at
that time. If the right hand side of the rule is empty, no states are popped
off of the stack: the uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions
and values. When a rule is reduced, the code supplied with the rule is
executed before the stack is adjusted. In addition to the stack holding the
states, another stack, running in parallel with it, holds the values returned
from the lexical analyzer and the actions. When a shift takes place, the value
assigned to yylex's parameter is copied onto the value stack. After the
return from the user code, the reduction is carried out. When the goto
action is done, the external variable yypval (i.e. '$$') is copied onto the value
stack. The pseudo-variables $1, $2, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The accept
action indicates that the entire input has been seen and that it matches the
specification. This action appears only when the lookahead token is the end­
marker, and indicates that the parser has successfully done its job. The error
action, on the other hand, represents a place where the parser can no
longer continue parsing according to the specification. The input tokens it
has seen, together with the lookahead token, cannot be followed by anything
that would result in a legal input. The parser reports an error, and attempts
to recover the situation and resume parsing: the error recovery (as opposed
to the detection of error) will be covered in Section 7.

It is time for an example! Consider the specification

```plaintext
%token ding dong dell
%
rhyme : sound place ;
sound : ding dong ;
place : dell ;
```

When Yacc is invoked with the '-l' option a description of the parser actions is
written to standard output. The description of the above grammar produced
by the '-l' option would be:

```plaintext
*** terminals ***
262 dell
260 ding
261 dong
0 end

*** non terminals ***
263 <system goal symbol>
265 sound

*** the productions ***
1 <system goal symbol> : end rhyme end
2 rhyme : sound place
3 sound : ding dong
4 place : dell

*** a vocabulary cross-reference ***
dell 4
ding 3
dong 3
```
end 1 1
<system goal symbol> -1
place 2 -4
rhyme 1 -2
sound 2 -3

*** the state sets ***

state: 1
  1 <system goal symbol> : . end rhyme end
  end
  the transitions:
    2

state: 2
  1 <system goal symbol> : end . rhyme end
  end
  the transitions:
    3 4 5

state: 3
  3 sound : ding . dong
dell
  the transitions:
    6

state: 4
  1 <system goal symbol> : end rhyme . end
  end
  the transitions:
    7

state: 5
  2 rhyme : sound . place
dell
  end
  the transitions:
    8 9

state: 6
  3 sound : ding dong .
dell
  the reductions:
    3 dell

state: 7
  1 <system goal symbol> : end rhyme end .
  end
  the reductions:
    1 end
The first section lists the grammar symbols and their definitions. The second section lists the grammar productions. Notice that there is an extra production added at the beginning of the grammar. This production makes it easy to start and stop the parser in a standard state. The "vocabulary cross-reference" list lists each of the grammar symbols and the numbers of the productions the symbol occurs in. The "-" sign in front of a production number indicates a definition production for the symbol. The last section describes the parser. Notice that, in addition to the actions for each state, there is a description of the parsing rules being processed in each state. The dot is used to separate what has been seen, and what is yet to come, in each rule. In fact, for each rule, the symbol in front of the dot is the lookahead symbol which must be recognized before shifting to the state containing that rule. Each rule in a state is directly followed by a list of terminals. These terminals make up the FOLLOW set for the LHS of the rule, i.e. the set of all terminals which can appear immediately to the right of A, where A is the LHS symbol of the rule. Each state also lists the possible transition states, and all rules which can be reduced in the state. The list of tokens following the reduction rule number are the set of lookahead terminals which would signal a reduction to this rule.

Suppose the input is

    ding dong dell

It is instructive to follow the steps of the parser while processing this input.

Initially, the current state is state 1. The parser provides the first token, the endtoken, for this special state. The action upon seeing end is to shift to state 2, so state 2 is pushed onto the stack and becomes the current state. The parser needs to refer to the input in order to decide between the actions available in state 2, so the first real token, ding, is read, becoming the lookahead token. State 2 has three transition states, 3, 4, and 5. The lookahead token for state 3 is ding (the symbol before the dot in the production in state 3). So, the action in state 2 on ding is is "shift 3", so state 3 is pushed onto the stack, and the lookahead token is cleared. State 3 becomes the current state. The next token, dong, is read, becoming the lookahead token. The
action in state 3 on the token *dong* is "shift 6", since the only transition state is 6 and the symbol before the dot is *dong* in state 6. So, state 6 is pushed onto the stack, and the lookahead is cleared. The stack now contains 1, 2, 3, and 6. In state 6, the only action is to reduce by rule 3.

sound : ding dong

This rule has two symbols on the right hand side, so two states, 6 and 3, are popped off of the stack, uncovering state 2. Consulting the description of state 2, there are again, three transition states. State 5 becomes the goto state, since the symbol before the dot is the same as the left hand side of the rule just reduced by. Thus state 5 is pushed onto the stack, becoming the current state.

In state 5, the next token, *dell*, must be read. The action is "shift 8", so state 8 is pushed onto the stack, which now has 1, 2, 5, and 8 on it, and the lookahead token is cleared. In state 8, the only action is to reduce by rule 4. This has one symbol on the right hand side, so one state, 8, is popped off, and state 5 is uncovered. The goto in state 5 on *place*, the left side of rule 4, is state 9. Now, the stack contains 1, 2, 5, and 9. In state 9, the only action is to reduce by rule 2. There are two symbols on the right, so the top two states are popped off, uncovering state 2. In state 2, there is a goto on *rhyme* causing the parser to enter and stack state 4. In state 4, the input is read; the endmarker is obtained, indicated by "end" in the description file. The action in state 4 when the endmarker is seen is to shift to state 7, so now the stack contains states 1, 2, 4, and 7. In state 7, the only action is to reduce by rule 1, popping 3 states off the stack, leaving state 1 on top. Although it is not indicated, the parser knows that state 7 is the final state, and whenever the final state is entered and the next lookahead token is the endmarker, the action is to accept, successfully ending the parse. The final state is always the state containing the added production rule fully parsed, (i.e. the dot is at the end of the extra production). successfully ending the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as "ding dong dong", "ding dong", "ding dong dell dell", etc. A few minutes spent with this and other simple examples will probably be repaid when problems arise in more complicated contexts.

5. Ambiguity and Conflicts

A set of grammar rules is *ambiguous* if there is some input string that can be structured in two or more different ways. For example, the grammar rule

\[ \text{expr} : \text{expr} \ '‐' \ \text{expr} \]

is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign
between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

\[ \text{expr - expr - expr} \]

the rule allows this input to be structured as either

\[ ( \text{expr - expr} ) - \text{expr} \]

or as

\[ \text{expr - ( expr - expr )} \]

(The first is called "left association", the second "right association").

Yacc detects such ambiguities when it is attempting to build the parser. It is instructive to consider the problem that confronts the parser when it is given an input such as

\[ \text{expr - expr - expr} \]

When the parser has read the second \text{expr}, the input that it has seen:

\[ \text{expr - expr} \]

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule. After applying the rule; the input is reduced to \text{expr} (the left side of the rule). The parser would then read the final part of the input:

\[ - \text{expr} \]

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, when the parser has seen

\[ \text{expr - expr} \]

it could defer the immediate application of the rule, and continue reading the input until it had seen

\[ \text{expr - expr - expr} \]

It could then apply the rule to the rightmost three symbols, reducing them to \text{expr} and leaving

\[ \text{expr - expr} \]

Now the rule can be reduced once more; the effect is to take the right associative interpretation. Thus, having read
the parser can do two legal things, a shift or a reduction, and has no way of
deciding between them. This is called a "shift/reduce conflict". It may also
happen that the parser has a choice of two legal reductions; this is called a
"reduce/reduce conflict". Note that there are never any "shift/shift"
conflicts.

When there are shift/reduce or reduce/reduce conflicts, Yacc still produces
a parser. It does this by selecting one of the valid steps wherever it has a
choice. A rule describing which choice to make in a given situation is called
a "disambiguating rule".

Yacc invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the earlier
grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred whenever there is a choice, in
favor of shifts. Rule 2 gives the user rather crude control over the behavior
of the parser in this situation, but reduce/reduce conflicts should be avoided
whenever possible.

Conflicts may arise because of mistakes in input or logic, or because the
grammar rules, while consistent, require a more complex parser than Yacc
can construct. The use of actions within rules can also cause conflicts, if the
action must be done before the parser can be sure which rule is being recog­
nized. In these cases, the application of disambiguating rules is inappropri­
ate, and leads to an incorrect parser. For this reason, Yacc always reports
the shift/reduce and reduce/reduce conflicts resolved by Rule 1 or Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a
correct parser, it is also possible to rewrite the grammar rules so that the
same inputs are read but there are no conflicts. For this reason, most previ­
ous parser generators have considered conflicts to be fatal errors. Further
experience suggests that this rewriting is somewhat unnatural, and produces
slower parsers; thus, Yacc will produce parsers even in the presence of
conflicts.

As an example of the power of disambiguating rules, consider a fragment
from a programming language involving an "if-then-else" construction:

\[
\text{stat} : \text{IF} \ ('\text{cond}') \text{ stat} \\
| \text{IF}('\text{cond}') \text{ stat ELSE stat} \\
\]

In these rules, IF and ELSE are tokens, cond is a nonterminal symbol
describing conditional (logical) expressions, and stat is a nonterminal symbol
describing statements. The first rule will be called the *simple-if* rule, and the second the *if-else* rule.

These two rules form an ambiguous construction, since input of the form

\[ \text{IF ( C1 ) IF ( C2 ) S1 ELSE S2} \]

can be structured according to these rules in two ways:

\[
\begin{align*}
\text{IF ( C1 )} \\
\{ \\
\text{IF ( C2 ) S1} \\
\} \\
\text{ELSE S2}
\end{align*}
\]

or

\[
\begin{align*}
\text{IF ( C1 )} \\
\{ \\
\text{IF ( C2 ) S1} \\
\text{ELSE S2} \\
\}
\end{align*}
\]

The second interpretation is the one given in most programming languages having this construct. Each *ELSE* is associated with the last preceding "un-ELSE'd" *IF*. In this example, consider the situation where the parser has seen

\[ \text{IF ( C1 ) IF ( C2 ) S1} \]

and is looking at the *ELSE*. It can immediately reduce by the simple-if rule to get

\[ \text{IF ( C1 ) stat} \]

and then read the remaining input,

\[ \text{ELSE S2} \]

and reduce

\[ \text{IF ( C1 ) stat ELSE S2} \]

by the if-else rule. This leads to the first of the above groupings of the input.

On the other hand, the *ELSE* may be shifted, *S2* read, and then the right hand portion of

\[ \text{IF ( C1 ) IF ( C2 ) S1 ELSE S2} \]

can be reduced by the if-else rule to get

\[ \text{IF ( C1 ) stat} \]

-17-
which can be reduced by the simple-if rule. This leads to the second of the above groupings of the input, which is usually desired.

Once again the parser can do two valid things - there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as

\[ \text{IF ( C1 ) IF ( C2 ) S1} \]

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of Yacc are best understood by examining the verbose (-v) option output file. For example, the output corresponding to the above conflict state might be: state: 23

18 stat: IF ( cond ) stat.
19 stat: IF ( cond ) stat. ELSE stat

the transitions:

45

the reductions:

18 ELSE *** WARNING: unresolved SHIFT/REDUCE conflict. ***
State, Rule, and Token involved are: 45, 18, ELSE *** Default action is to SHIFT.

The last three lines following the state description describe the conflict, giving the state, rule, and the input symbol. Recall that the dot marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to

\[ \text{IF ( cond ) stat} \]

and the two grammar rules shown are active at this time. The parser can do two possible things. If the input symbol is ELSE, it is possible to shift into state 45. State 45 will have, as part of its description, the line

19 stat: IF ( cond ) stat ELSE . stat

since the ELSE will have been shifted in this state. Back in state 23, the other action is to reduce by grammar rule 18. By default, the shift will be done, if the lookahead token is ELSE. Otherwise, the parser will reduce by grammar rule 18:

18 stat: IF '(' cond ')' stat

Once again, notice that the numbers following "shift" commands refer to other states, while the numbers following "reduce" commands refer to
grammar rule numbers. In most states, there will be at most one reduce action possible in the state. The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate. In really tough cases, the user might need to know more about the behavior and construction of the parser than can be covered here. In this case, one of the theoretical references [1,2,3] might be consulted; the services of a local guru might also be appropriate.

6. Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient; this is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the form

```
expr : expr OP expr
```

and

```
expr : UNARY expr
```

for all binary and unary operators desired. This creates a very ambiguous grammar, with many parsing conflicts. As disambiguating rules, the user specifies the precedence, or binding strength, of all the operators, and the associativity of the binary operators. This information is sufficient to allow Yacc to resolve the parsing conflicts in accordance with these rules, and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a Yacc keyword: %left, %right, or %nonassoc, followed by a list of tokens. All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus,

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative, and have lower precedence than star and slash, which are also left associative. The keyword %right is used to describe right associative operators, and the keyword %nonassoc is used to describe
operators, like the operator .LT. in Fortran, that may not associate with themselves; thus,

\[ A \ .LT. \ B \ .LT. \ C \]

is illegal in Fortran, and such an operator would be described with the keyword %nonassoc in Yacc. As an example of the behavior of these declarations, the description

\[
\begin{align*}
%right & \ '=' \\
%left & \ '+' \ '-' \\
%left & \ '*' \ '/' \\
\%
\end{align*}
\]

\[
\text{expr : expr } '=' \text{ expr} \\
\text{ | expr } '+' \text{ expr} \\
\text{ | expr } '-' \text{ expr} \\
\text{ | expr } '*' \text{ expr} \\
\text{ | expr } '/' \text{ expr} \\
\text{ | NAME}
\]

might be used to structure the input

\[
a = b = c \cdot d - e - f \cdot g
\]
as follows:

\[
a = \left( b = \left( \left( (c \cdot d) - e \right) - (f \cdot g) \right) \right)
\]

A token declared by %left, %right, and %nonassoc should not be declared by %token as well. The token definitions can be declared on these lines the same way as on %token lines. A token declared with %token, has no associativity and no precedence associated with it.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those terminals that have them.

2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last terminal in the body of the rule. If the rule contains no terminals, or if the last terminal has no precedence, the rule will have no precedence or associativity associated with it.

3. When there is a reduce/reduce conflict and either of the the grammar rules in conflict has no associated precedence, or there is a
shift/reduce conflict and either the input symbol or the grammar rule has no associated precedence, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.

4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

7. Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found; it is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string, and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, Yacc provides a simple, but reasonably general, feature. The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current lookahead token, and performs the action encountered. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted. A skeleton error message routine, yyerror, is provided in the parse library. This routine is called when an error is first detected. The user may supply a version of yyerror. The argument to yyerror is a string containing an error message. The default version of yyerror prints error messages like "syntax error at line 50". The average application will want to do better than this.

As an example, a rule of the form

```plaintext
stat : error
    ;
```
would, in effect, mean that on a syntax error the parser would attempt to skip over the statement in which the error was seen.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general, but difficult to control. Somewhat easier are rules such as

```
stat : error ';
```

Here, when there is an error, the parser attempts to skip over the statement, but will do so by skipping to the next ';'. All tokens after the error and before the next ';' cannot be shifted, and are discarded. When the ';' is seen, this rule will be reduced, and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications, where it may be desirable to permit a line to be reentered after an error. A possible error rule might be

```
input : error '@n'
   %{
      call printf("Reenter last line:")
   }
```

There is one potential difficulty with this approach; the parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens, and gives no message; this is clearly unacceptable. For this reason, there is a macro that can be used to force the parser to believe that an error has been fully recovered from. The statement

```
YYERROK
```

in an action resets the parser to it's normal (non-error-recovery) mode. The last example is better written as

```
input : error '@n'
   %{
      YERROK
      call printf("Reenter last line:")
   }
```

As mentioned above, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate; for example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous
lookahead token must be cleared. The macro statement

YYCLEARIN

in an action will have this effect. For example, suppose the action after error were to call some sophisticated resynchronization routine, supplied by the user, that attempted to advance the input to the beginning of the next valid statement. After this routine was called, the next token returned by yylex would presumably be the first token in a legal statement; the old illegal token must be discarded, and the error state reset. This could be done by a rule like:

```
stat : error
    %{ 
    call resynch
    YYERROR
    YYCLEARIN
    %} ;
```

NOTE: the user must declare the function yylex in the declarations section in order to use YYCLEARIN.

These mechanism are admittedly crude, but do allow for a simple, fairly effective recovery of the parser from many errors; moreover, the user can get control to deal with the the error actions required by other portions of the program.

8. The Yacc Environment

When the user inputs a specification to Yacc, a rat4 program is written to standard output. The Yacc parse function is called *yyparse* and it is an integer valued function with an integer valued argument. When it is called, it in turn repeatedly calls *yylex*, the lexical analyzer supplied by the user (see Section 3) to obtain input tokens. Eventually, either an error is detected, in which case (if no error recovery is possible) *yyparse* returns (and sets it's argument to) the value OK, or the lexical analyzer returns the endmarker token and the parser accepts. In this case, *yyparse* returns (and sets it's argument to) the value ERR. This rat4 program generated by Yacc must be compiled with the rat4 compiler and linked with the library *yylib*.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, a main program should be supplied, that eventually calls *yyparse*. In addition, a routine called *yyerror* prints a message when a syntax error is detected. This routine must be supplied in one form or another by the user. To ease the initial effort of using Yacc, a default version of yyerror is provided in the parse library *yylib*. The usual place for the main program, yyerror, and yylex is in the *programs* section of the yacc input. (See section 1). Here is a trivial example of a main
program and a yyerror routine.

```c
### main - main program to call the parser
#
DRIVER(main)

  integer yyparse
  integer sts

  if ( yyparse(sts) == ERR )
    call error( "main: FATAL ERROR parsing input." )

DRETURN
end

### yyerror - print error message
#
subroutine yyerror( messag )
character messag(ARB)

  call fprintf(ERROUT, "Yacc parser error: %s@N", messag)

return
end
```

The '-d' flag in yacc puts the generated parser in debug mode. The parser will output a verbose description of its actions, including a discussion of which input symbols have been read, and what the parser actions are.

9. Hints For Preparing Specifications

This section contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

9.1. Lexical Tie-ins

Yacc provides a way for the user to combine a lexical scanner specified with Lex and a parser specified with Yacc. The user can simply place the Lex source in the programs section of the Yacc input. Yacc just copies this section verbatim. Yacc also outputs rat4 definitions for each of the tokens specified in it's declarations section so that the token names can be used in the Lex specification. For example

```
%token WORD 23 NUMBER 34
```

would produce the rat4 definitions

```c
define(WORD,23)
define(NUMBER,34)
```
Yacc produces a rat4 program with the programs section first (copied verbatim), followed by the generated semantic routine and tables. Thus Yacc could produce Lex source code, if the Yacc programs section contained a Lex source (the semantics and parse tables would just become a part of the programs section of the Lex source). The output of Yacc could be processed by Lex to create a combined parser-scanner rat4 program.

10. Advanced Topics

To be added later.

11. References


Appendix A: A Simple Example

To be added later.

Appendix B: An Advanced Example

To be added later.

FILES

none

SEE ALSO

yacc(tool), lex(tool), yaclr(tool), rc(tool), lrgen(tool), pr(tool), yyplib(lib)
"Yacc: Yet Another Compiler-Compiler" by S. C. Johnson
"LR - Automatic Parser Generator and LR(1) Parser" by C. Wetherell and A. Shannon.

AUTHOR

Major portions of this writeup were adapted from S. C. Johnson's "Yacc: Yet Another Compiler-Compiler". Description's of RTSG peculiarities were added by Theresa Breckon.
NAME
adventure - The game of adventure

SYNOPSIS
adventure

DESCRIPTION
Adventure is a fantasy game from DECUS.

"Somewhere nearby is Colossal Cave, where others have found fortunes in treasure and gold, though it is rumored that some who enter are never seen again. Magic is said to work in the cave. I will be your eyes and hands. Direct me with commands of 1 or 2 words. I should warn you that I look at only the first four letters of each word, so you'll have to enter 'NOR THEAST' as 'NE' to distinguish it from 'NORTH'. (Should you get stuck, type 'HELP' for some general hints. For information on how to end your adventure, etc., type 'INFO')."

The primary changes incorporated in Release 3 are the following:

- Improved page formatting.
- Upper and lower case output.
- SAVE and RESTORE.
- Fast initialization feature.

SAVE (synonyms SUSPEND and PAUSE) permits the user to save the current state of the game for later analysis or continuation. RESTORE (synonym RESUME) restores a previously SAVEd game. The fast initialization feature reduces the startup time of the game from 60-90 seconds to 3-5 seconds after the first game.

FILES
+LIB/aindx.dat - program data files
+LIB/atext.dat
atext.txt - text data file (used to create atext.dat)
asave.dat - saved game file

DIAGNOSTICS
The game provides very friendly diagnostics.

AUTHOR
Willie Crowther, Don Woods (don@SU-AI), Kent Blackett, Bob Supnik.
NAME
blast - visual Muzak generator

SYNOPSIS
blast

DESCRIPTION
Blast is a simulation of the constantly competing processes of osteoblasis and osteoplasis. It is reasonably pretty to watch.

SEE ALSO
cardio(tool), kalid(tool)

AUTHOR
Jef Poskanzer

BUGS/DEFICIENCIES
Not useful. Screws up on VT100's. Can cause squeamish people to get ill if they know what they are watching.
NAME
boxes - draw boxes

SYNOPSIS
boxes [-i <i>] [-n <n>] [-r <r>]

DESCRIPTION
Draws boxes on the screen. As boxes grow old, they are replaced by new ones. <i> new boxes will be added at a time. By default, 1 box is added at a time.

This process of replacing old boxes by new ones is repeated <r> times. The default for <r> is infinity.

There will be a maximum of <n> boxes on the screen at any one time. The default number of boxes is 30.

Thus, to just draw 25 boxes all at once and exit, do

% boxes -i 25 -n 25 -r 1

AUTHOR
Todd Hammond

BUGS/DEFICIENCIES
The boxes tend to congregate at the bottom right corner of the screen. Although this is deliberate, some people might prefer another distribution of sizes and positions.
NAME
cardio - electrocardiogram simulation

SYNOPSIS
cardio

DESCRIPTION
Cardio displays a simulated electrocardiogram on the screen. It is reasonably pretty to watch.

It gets the terminal type from the TERM environment variable.

SEE ALSO
kalid(tool), blast(tool)

AUTHOR
Jef Poskanzer

BUGS/DEFICIENCIES
Not useful.
NAME
chown - change file owner

SYNOPSIS
chown username file ... chown [grp,mem] file ...

DESCRIPTION
Chown is used to change the ownership of files and directories. Normally, the
new file owner is specified by a user name, but an octal uic string may also
be specified, e.g.:

% chown [231,4] /mnt/leres/login.com

SEE ALSO
chmod(tool)

DIAGNOSTICS
"chown: Not a privileged user."

This error message is generated when you don't have the privileges neces­
sary to change the ownership of the file.

"chown: Invalid uic string format."

This error message indicates that a uic string was incorrectly specified.

AUTHOR
Craig Leres
NAME
  clear - clear the screen

SYNOPSIS
  clear

DESCRIPTION
  Clear will clear the terminal's screen.

SEE ALSO
  sclb(lib)

AUTHOR
  Jef Poskanzer
NAME
crembx - create permanent vms mailbox

SYNOPSIS
crembx [-b <number>] [-m <number>]
[-s[rwed]] [-o[rwed]] [-g[rwed]] [-w[rwed]] name

DESCRIPTION
Crembx is used to create permanent vmx mailboxes. The privilege PRMMBX
is required for use.

Here is a list of the flags that crembx accepts:

-b <number> Specify the size of the mailbox buffer. By default, the mailbox buffer is
80 bytes long.

-m <number> Specify the number of mailbox buffers that the system will store. The
default is 5 buffers.

The following flags are used to specify the protections on the new mailbox.
The default is to grant all permissions to all classes of users.

-s Specify the system class protections.

-o Specify the owner class protections.

-g Specify the group class protections.

-w Specify the world class protections.

Each class flag may be followed by some combination of the letters "rwed", to
give the specified class of users access to the mailbox. The letters "rwed" stand for read, write, execute and delete. User classes that are not specified
are granted all permissions.

Since the mailbox name specified becomes a system wide logical name,
crembx always converts the name to uppercase letters. It is STRONGLY sug-
gested that a ' $' or ' _' character is included in the name to prevent logical
name conflicts for users on the system.

SEE ALSO
delmbx(tool)
NAME
   delmbx - delete permanent vmx mailbox

SYNOPSIS
   delmbx name

DESCRIPTION
   Delmbx is used to delete permanent vmx mailboxes created with the crembx
tool. The privilege PRMMBX is required for use.

SEE ALSO
   crembx(tool)

AUTHOR
   Craig Leres

BUGS/DEFICIENCIES
   For some reason, the sys$delmbx call cannot delete mailboxes that have a
   logical name that includes lowercase characters. To remove such mailboxes,
   specify the actual mailbox name, rather than the mailbox's logical name.
NAME

des - encrypt or decrypt using the Data Encryption Standard

SYNOPSIS

des [-e/-d] key

DESCRIPTION

Des encrypts or decrypts characters on standard input using the National
Bureau of Standards' Data Encryption Standard. It uses deslb, a portable
implementation of the standard. It is a fairly slow implementation: 65 msec
per 64-bit block, or 8 msec per character. Thus, encrypting a 5000-
character message would take 40 seconds.

The -e flag (or no flag) specifies encryption; -d specifies decryption. Multiple
encryption (encrypting a file with first one key and then another) is allowed,
and in fact recommended for high-security applications, but the decryption
must be done in the exact reverse order as encryption was done.

The key string is converted into a 64-bit DES key by taking each character
and XORing the low-order 7 bits into successive 7-bit slots in the key, treated
circularly. Thus, for the 10th character of the string, bit 1 is dropped, bit 2
is XORed into bit 64 of the key, and bits 3-8 are XORed into bits 1-6 of the
key. After all the characters have been processed, bits 8, 16, ..., 64 are clob-
ered so that each byte has odd parity. This imposition of odd parity is
required by the standard.

des does cipher-block chaining, in which each 64-bit block of cleartext is first
XORed with the ciphertext of the preceding block, before being passed to
the DES encryption. For the first block, since there is no preceding block,
the key is used.

After encryption, each 64-bit block gets broken up into 10 6-bit chunks plus a
4-bit nybble. des then adds 32 to each chunk and writes it out as a charac-
ter. Furthermore, each block of 64 bits => 11 characters is terminated with
a newline. This output processing makes the ciphertext mailable. The price,
of course, is that the ciphertext is 1.5 times larger than the cleartext.

SEE ALSO

deslb(lib), crypt(tool), cyplb(lib)

AUTHOR

Jef Poskanzer
NAME
dungeon - the game of Zork

SYNOPSIS
dungeon

DESCRIPTION
Dungeon is a game of adventure, danger, and low cunning. In it you will explore some of the most amazing territory ever seen by mortal man. Hardened adventurers have run screaming from the terrors contained within.

On-line information may be obtained with the commands HELP and INFO.

FILES
+LIB/dindx.dat - initialization data base
+LIB/dtext.dat - main data base
dsave.dat - saved game data file

AUTHOR
Dungeon was created at the Programming Technology Division of the MIT Laboratory for Computer Science by Tim Anderson, Marc Blank, Bruce Daniels, and Dave Lebling. It was inspired by the Adventure game of Crowther and Woods, and the Dungeons and Dragons game of Gygax and Arneson. The original version was written in MDL (alias MUDDLE). The current version was translated from MDL into FORTRAN IV by a somewhat paranoid DEC engineer who prefers to remain anonymous.
NAME
empire - the wargame of the century

SYNOPSIS
empire

DESCRIPTION
Empire is a simulation of a full-scale war between two emperors, the computer and you. Naturally, there is only room for one, so the object of the game is to destroy the other. The computer plays by the same rules that you do.

1. Introduction

The map is a rectangle 600*1000 miles on a side. The resolution is 10, so the map you see is 60*100. The map consists of sea='.', land='+', Uncontrolled cities='*', Computer-controlled cities='X', and Your dominated cities='0'. Each emperor gets 1 move per round (1 round=1 day), moves are done sequentially.

The map is displayed on the player's screen during movement. Each piece is represented by a unique character on the map. With a few exceptions, you can only have ONE piece on a given location. On the map, you are shown only the 8 squares adjacent to your units. This information is updated before and after every move. The map displays the most recent information known.

The game starts by assigning you one city and the computer one city. Cities can produce new units. Every city that you own produces more pieces for you according to the cost of the desired unit. The typical play of the game is to issue the Automove command until you decide to do something special. During movement in each round, the player is prompted to move each piece that does not otherwise have an assigned function.

Map coordinates are 4-digit numbers. the first two digits are the row, the second two digits are the column.

The pieces are as follows:

<table>
<thead>
<tr>
<th>Piece</th>
<th>Yours</th>
<th>Enemy</th>
<th>Moves</th>
<th>Hits</th>
<th>Cost</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Army</td>
<td>A</td>
<td>a</td>
<td>1</td>
<td>1</td>
<td>5(6)</td>
<td>500</td>
</tr>
<tr>
<td>Fighter</td>
<td>F</td>
<td>f</td>
<td>4</td>
<td>1</td>
<td>10(12)</td>
<td>200</td>
</tr>
<tr>
<td>Destroyer</td>
<td>D</td>
<td>d</td>
<td>2</td>
<td>3</td>
<td>20(24)</td>
<td>200</td>
</tr>
<tr>
<td>Submarine</td>
<td>S</td>
<td>s</td>
<td>2</td>
<td>2</td>
<td>25(30)</td>
<td>200</td>
</tr>
<tr>
<td>Troop transport</td>
<td>T</td>
<td>t</td>
<td>2</td>
<td>3</td>
<td>30(36)</td>
<td>100</td>
</tr>
<tr>
<td>Cruiser</td>
<td>R</td>
<td>r</td>
<td>2</td>
<td>8</td>
<td>50(60)</td>
<td>100</td>
</tr>
<tr>
<td>Aircraft carrier</td>
<td>C</td>
<td>c</td>
<td>2</td>
<td>8</td>
<td>60(72)</td>
<td>100</td>
</tr>
<tr>
<td>Battleship</td>
<td>B</td>
<td>b</td>
<td>2</td>
<td>12</td>
<td>75(90)</td>
<td>100</td>
</tr>
</tbody>
</table>
The second column shows the map representation for your units.

The third shows the representations of enemy units.

Moves is the number of squares that a unit can move in a single round.

Hits is the amount of damage a unit can take before it is destroyed.

Cost is the number of rounds needed to produce another of the same unit.

The number in parenthesis is the cost for a city to produce the 1st unit.

The last column is the maximum number of that unit that you can have.

2. Description of the pieces

Army: Armies move only on land. ONLY ARMIES CAN CAPTURE CITIES. They have a 50% probability of doing so. Attacking one's own cities results in the army’s destruction. Armies can be carried by troop transports. Just move the army on the transport and when the transport moves the army moves with it. You cannot attack any ships while on board a transport. YOU CANNOT MOVE BACK ON A CITY WITH AN ARMY.

Fighter: Fighters move over sea or land. They move 4 times per round. They are refueled at controlled cities and carriers. They are shot down over uncontrolled cities. They have a max range of 20 spaces.

Ships, general: All ships can move only on the sea. They move two times per round. Ships can also dock in a controlled city. Docked ships have damage repaired at rate of 1 hit per day. If a ship is hit badly, it will slow to 1 move per round.

Destroyer: Typical ship, quickest to produce.

Submarine: When a submarine scores a hit, 3 hits are exacted instead of the usual 1 from the enemy unit. This is the only unit having this property.

Troop Transport: Troop Transports are the only units that can carry armies. They can carry a maximum of 2 * (the number of hits left) of armies. Armies that cannot be carried will drown.

Cruisers: Typical ship.

Aircraft Carriers: Carriers are the only ships that can carry fighters. Carriers carry a maximum of the number of hits left of fighters.

Battleship: Typical ship.
3. **Functions to which you can assign your pieces** Other than just moving your pieces you can assign them the following automatic functions:

- **awake**: Cancel current automatic function and return to manual moves.
- **sentry**: Stay put, do not ask the player to move the piece, wake up if an enemy piece comes within sensor range.
- **direction**: Move in specified direction, wake up if an enemy piece, enemy city, or unoccupied city is encountered. Temporary wake up if an obstacle is in path of movement, after getting a manual move from you, THE UNIT IS STILL ASSIGNED A DIRECTION. A direction assignment is represented by the key which sets that direction (e.g.: D means east).
- **move**: Move towards location assigned to the piece (in editing mode). Wake up if enemy piece is encountered. Wake up temporarily if obstacle is in path of movement. Represented by the coordinate the piece is moving toward.
- **fill**: (Troop transports and aircraft carriers only) go on sentry duty until full to capacity of armies or fighters.
- **random**: (Armies only) move at random subject to the following conditions:
  - If an uncontrolled city is adjacent, attack it.
  - If an enemy unit is adjacent, attack it (even if it is a ship).
  - If an unfilled troop transport of yours is adjacent, get on it and wake up.
  - Move if possible without attacking any of your own units.
  - It will not destroy itself unless it is in a city surrounded by your units.

4. **Orders mode** The top level prompt is:

- **Your Orders?**

   This is asked between each round (if you are not in Auto move mode). The following commands are valid at this time:

   - **A**: Auto move. Begin movement, stay there until "0" in move mode cancels the auto move.
   - **C**: Give the computer a free move.
   - **H**: Display the Help screen. Contains a brief description of all the commands.
   - **J**: Puts you into Editing Mode (explained later), where you can examine and/or change the functions associated with your pieces and cities.
   - **M**: Move. Cause a round to be played by you and the computer.
N: Give the computer the number of free moves you specify. The game gets more interesting if you give the computer 10 to 30 free moves at the start.

P: Re-display current sector on screen.

R: Display the round number.

S: Clears the screen.

T: Request a printout of the entire map. You must supply a file spec for where you want the map put.

Q: Quit the game. Be sure to save first. ip V:.8 Save game.

5. **Movement mode**  To simply move a piece, type one of the following keys:

```
Q W E
/   
A--+--D
| Z X C
```

These keys move in the direction of the key from S. The characters are not echoed and only 1 character is accepted, so no need for a <Return>. Hit the SPACE BAR if you want the piece to stay put.

Other commands are:

H: Display Help text (hit any character to continue moving)

J: Enter Editing Mode

G: Fill: put the troop transport (or aircraft carrier) to sleep until it accumulates 6 armies (or 8 fighters), then wake it up. If the ship is damaged, the ship will wake up when it has all it can take.

I: Set unit to moving in a direction specified by the next character typed in

K: Wake up piece. If piece is a troop transport or carrier, all armies or fighters on board are also woken up.

L: Set fighter path for city to be the direction following the "L".

O: Cancel auto move. At the end of the round, Orders Mode will prompt. Doesn’t affect current piece.

P: Refresh the screen

R: If it’s an army, set it to moving at random.
S: Put on sentry duty.

?: Display information about the piece. Shows the function, hits left, range and number of armies or fighters aboard.

ATTACKING something is accomplished by moving onto the square of the unit you wish to attack. Hits are traded off at 50% probability of a hit landing on one or the other units until one unit is totally destroyed. There is only 1 possible winner.

If you give a piece a direction or move function, they will wake TEMPORARILY if they run into an obstacle (or enemy). You must explicitly wake the piece to regain complete control or assign it a new function.

Fighters moving under the command of a function will wake up when they have 10 rounds of fuel left. This is to enable you to decide whether you want to make it kamikaze or send it back to a city for refueling. Be careful to cancel any currently assigned function before trying to bring the fighter back.

You are "allowed" to do FATAL things like, attack your own cities or other pieces. If you try to do fatal move that involve terrain restrictions, like, drive armys into the sea and ships into land, you are given a chance to reconsider. Answer with an "n" if you want to commit suicide. You cannot move onto the edge of the world.

6. Editing mode Editing mode allows you to move around the "world" and check on things. You can assign and deassign movements and inquire on the production of cities. Movements assigned during editing mode do not take effect until the next round.

To move the cursor around, use the standard direction keys.

Other commands are:

H: Display Help text (hit any character to continue editing).

O: Exit from editing mode.

I: Give piece (or city) the function 'direction', enter the key specifying the direction following the 'I'.

K: Wake up piece (or cancel city fighter path).

M: Put piece (or city) in 'move' function. Type 'M' over piece (or city), then move the cursor to where you want it to go, and type 'N'. Assigning a 'move' to a city, effects any fighters that land there.

N: Specify the end point of a move (see M command).
P: Display new sector. Each sector represents a 20*70 area of the map, arranged as follows:

\[
\begin{array}{cccc}
0 & 5 \\
1 & 6 \\
2 & 7 \\
3 & 8 \\
4 & 9 \\
\end{array}
\]

The sectors overlap by 8 vertically, and 40 horizontally.

R: Put army in 'random'.

S: Put piece in Sentry mode.

Y: Change phase of city that cursor is on top of. When program asks for production demands, key in the letter corresponding to what you want produced.

?: Display information about piece or city. For pieces, displays function, range, hits left, any pieces aboard. Cities display production, fighter paths, and any pieces in the city.

You can give cities functions. This doesn't affect the city any, but any fighter landing in that city will pick up the specified function. This is useful for setting up automatic fighter movements.

Note that you cannot affect anything inside a city with the editor.

AUTHOR

Mario DeNobili and Thomas N. Paulson.
Support for different terminal types added by Craig Leres.
NAME
film - motion picture generator

SYNOPSIS
film <filmfile>

DESCRIPTION
Film will display a short motion picture on your terminal. Some of the movies currently available are Duel, Blob, Bambi, Tree, and MrBill. It is fairly easy to construct your own films, too.

The film files are found using the FILM search path (which usually contains "/misc/films").

FILES
/misc/films - film directory

AUTHOR
Jef Poskanzer

BUGS/DEFICIENCIES
Not useful.
NAME
generate - random sentence generator

SYNOPSIS
generate @[bnffile@] @[-repcount@]

DESCRIPTION

Generate reads a BNF grammar and generates random sentences from it.
The syntax of the BNF files is somewhat similar to that used by yacc(tool).
For example, here is a simple BNF that generates random phone numbers:

phonenum: ( <area> <exch><number> ;
area: <notZeroOrOne><zeroOrOne><notZeroOrOne> ;
exch: <notZeroOrOne><digit><digit> ;
number: <digit><digit><digit><digit> ;
digit: 0|1|2|3|4|5|6|7|8|9;
notZeroOrOne: 2|3|4|5|6|7|8|9;
zeroOrOne: 0|1;

BNF files are opened using the BNF search path. If no BNF file is specified,
generate will read from STDIN. The repcount is a number saying how many
sentences to generate. If no repcount is specified, only one sentence will be
generated.

There are a few special names that are useful for generating nested indented
structures, for example Pascal source code:

<CR> Generate a carriage return / line feed.
<TAB> Generate blanks out to the current tabulation level.
<+TAB> Increase current tabulation level.
<-TAB> Decrease current tabulation level.

Lastly, generate will process include statements similar in form to rat4’s:

include "tabextras"

Include files are also opened with the BNF search path.

SEE ALSO
yacc(tool), lex(tool)
It's kind of slow.
NAME
  kalid - visual Muzak generator

SYNOPSIS
  kalid

DESCRIPTION
  Kalid displays a kalidioscope-like pattern on the screen. It is reasonably pretty to watch.

SEE ALSO
  cardio(tool), blast(tool)

AUTHOR
  Jef Poskanzer

BUGS/DEFICIENCIES
  Not useful.
NAME

man - display the manual entries for the tools specified

SYNOPSIS

man [valid more flags] toolname ...

DESCRIPTION

Man is a tool which locates and prints (onto standard output) a manual entry
describing the use of the specified tool(s).

If standard output is a terminal, the output is filtered with the more tool; i.e.
man invokes more with the "-s" flag, which causes the compression of multi-
ple newlines to a single newline. If the output of man isn't a terminal (in the
case of redirection or a pipe) then the output is not compressed.

The search path specified in the environment variable MAN is used to find the
desired manual entries.

Man passes flags through to more.

SEE ALSO

paths(info), more(tool)

AUTHOR

Craig Leres

BUGS/DEFICIENCIES

Man uses the +MAN search path to find a manual entry, not the +PATH
search path. Therefore, the manual entry found by man may not correspond
to the tool by the same name found by the shell. The user should keep his
+MAN and +PATH files compatible.
NAME
more - filter for crt viewing

SYNOPSIS
more [-<n>] [-cls] [file] ...
   -<n>: set window size to <n>
   -c: clear screen when drawing new screenfuls
   -l: display with line numbers
   -s: compress adjacent blank lines (squeeze)

DESCRIPTION
More reads each file (if none specified, the standard input) and writes it onto
the user's terminal, pausing for a display command after each screenful. If
the user types a space, the next screenful will be displayed. If a carriage
return is typed, more will write out the next line of the file. Other display
commands are discussed below. The various flags are:

-<n> sets the screen size to <n>. For example, the argument "-40" will set
the screen size to 40 lines. The screen size defaults to the number of
lines the system thinks your terminal has.

-c whenever more displays a new page of a file it should first clear the
screen. (Ignored if the terminal cannot clear the screen.)

-l display line numbers. Line numbers are written right-justified in the
first four columns on a line. A blank separates the line number from
the remainder of the line.

-s multiple, adjacent blank lines are displayed as one blank line.

A file name of "-" is taken to mean the standard input.

Before using more, setenv should be used to set the environment variable
TERM to make the user's terminal type available to more. If more is unable
to get information about the terminal specified in the TERM variable, more
will use characteristics of prototype "dumb" terminal.

A number of display commands can be given when more prompts after hav­
ing displayed a screenful of text. Many of the following commands can be
preceded by an optionally signed number. If the number is preceded by
either "+" or "-", it is taken to specify motion relative to the current posi­
tion, with "+" indicating forward motion and "-" indicating backward motion.
(E.g., "-5<space>" means "display the 5th screenful before the present one").
If the number is unsigned, it is taken to signify absolute motion. (E.g.,
"5<space>" means "display the 5th screenful of the file"). If no numeric
value is given, then a value of 1 is assumed for the number. (E.g., "-f" means
"display the previous file").

- 1 -
All commands except those taking strings as arguments are executed as soon as the command character is typed (i.e., commands do not have to be terminated with a carriage return). When the user types in these commands, the command characters are not shown on the screen. Commands which do take a string as an argument ("/", " a carriage return. When entering the string, control-U and control-X may be used to delete the characters which have been typed so far, control-W deletes the last word typed, and control-H and the Delete key delete the last character typed. If the user deletes (using control-H or Delete) past the prompt the command is aborted. Control-V may be used to escape any of these characters so that they lose their special meaning. Command characters may be typed in either lower or upper case. The display commands are:

<space> Display a screenful of text. A number can be given to specify which screenful of the file should be displayed. Default (if no number is given) is to display the next screenful (equivalent to "+1<space>").

<return> Display a screenful of text with the specified line written two lines from the top of the screen. The default is to display the next line of the file.

f If a number is given preceding this command, more begins displaying the specified file. If no number is given, more lists all of the files being processed.

<pattern> Search forward for the pattern <pattern>. A positive number may precede the command to indicate how many times the pattern should be found. For example, "5/foo" will find the 5th occurrence of the word "foo" following the current position in the file. The forward search starts at the third line from the top of the screen. Pattern may be any search pattern which can be used by ed(tool). If no pattern is specified, the pattern used for the last search command will be used.

<pattern> Search backward for the pattern <pattern>. The backward search starts at the line preceding the one displayed at the top of the screen.

n Repeat the last search the specified number of times (default: 1). If a negative number is given with this command, the search is made in the opposite direction.

% Display the first screenful of the file.

$ Display the last screenful of the file.

mc Mark the current file position with the character c. c may be any character except "?" or "". The return-to-marker command (see
below) may then be used later to return to this position. If c is a question mark ("?") a summary of all the markers will be displayed.

'c

Return to the specified marker. Returns to the file and line that were being displayed when the specified marker was set. If c is a single quote (""') then more will return to the file and line it was at before the last "jump" (either a backward move, a move to a new file, or a forward move of more than one screenful). If c is a question mark ("??") a summary of all of the markers will be displayed.

r<filename> Read in and display the file named <filename>. Any number given with this command specifies where in the current list of files the new file should be inserted (default: insert after the current file). If the number is 0, the new file will be inserted at the beginning of the current list. If the number is signed, the new file is inserted relative to the current file. If the number is unsigned, the new file is inserted after the file at the given position.

!<command> Spawn to a shell to execute <command>. If no command is given, more spawns an interactive shell.

= Display the current file name, file number, and line number.

w If a positive number is specified, it becomes the new window (screen) size. The maximum window size is 100. If no number is given, the value of the current window size is displayed. Otherwise, the value of the new window size is displayed.

c Toggle the setting of the -c (clear) option. (Ignored if the terminal cannot clear the screen.)

l Toggle the setting of the -l (line number) option.

s Toggle the setting of the -s (squeeze) option.

~L A control-L causes the screen to be cleared (if possible) and redrawn.

h Display the more help file.

? Same as the "h" command.

q Quit. Terminates the execution of more. Typing ~C, ~Y, or an end-of-file will also quit.

If a command character is typed which more does not recognize, the terminal's bell (if present) rings. If a command character is typed while
more is writing out a screenful of text, more will finish writing out the line it is currently displaying and then (without finishing the entire screen) execute the command. If any character is typed while more is searching through a file, the user is told how far the search has proceeded, the search is terminated, and the user is prompted for another command. The file remains positioned at the location where the search command was given, and the single quote ('"') marker is set at the position where the search terminated.

More tries to write out the "line of interest" (the line number specified before a <return> command, or the line containing the string which matched a search's pattern) two lines down from the top of the screen. If doing so, however, would result in less than a screenful of text being displayed (because the "line of interest" is near the end of the file), more gives higher priority to displaying a full screen of text.

More will inform the user if the end of a file is reached while displaying the file. Any forward motion, such as a <space> command to display the next screenful, will apply to the next input file (thus <space> will cause the first screenful of the next file to be displayed). If more has reached the end of the last input file and the user specifies forward motion, more terminates.

If the output of more is not a terminal, more simply copies the input files to the output. In this case, none of the options such as line-numbering (-l) or squeezing (-s) apply.

FILES
+MAN/morehelp - the more help file.
+LIB/termcap - terminal capabilities data base.

SEE ALSO
cat(tool), crt(tool), ed(tool), setenv(tool), UNIX more(1) and vi(1).

DIAGNOSTICS
more prints error messages if it can't open a file, and aborts if it can't open any of its input files.

If more's output is going to a terminal, then it will generate an error message if any of its input files is a terminal.

AUTHOR
Vern Paxson

BUGS/DEFICIENCIES
More cannot handle non-random access files which are not terminals (e.g., NLA0: on the Vax), and does not recover gracefully when given one as input.
NAME
plane - airplane flight simulation

SYNOPSIS
plane

DESCRIPTION
Plane provides a pseudo-graphic video presentation of a pilot's instrument panel in real-time.

The following keys are your controls:

'k' or 'w' - Nose up
'j' or 'x' - Nose down
'l' or 'd' - Increase power
'h' or 'a' - Decrease power
'1' or '8' - Turn left
'2' or '9' - Fly straight
'3' or '0' - Turn right
'-' - Redraw the screen
'Z' - Bail out

They may be depressed repeatedly for gross changes.

When the game starts, you will flying level at 2500 feet. Instructions from the ground radar controller will appear in the upper right hand corner of the screen. S/he will attempt to talk you down safely.

Note the following:

(1) The plane will stall below 75 mph.
(2) Touchdown above 100 mph results in a bounce.
(3) Descent rate must be below -10 fps at touchdown.
(4) A 100 foot hill is at the far end of the runway.
(5) The fuel supply is marginal.
(6) Not flying the correct heading wastes fuel.
(7) A forced landing is possible if touchdown is gentle.

AUTHOR
Originally written by Bill Greene, this version was hacked together by Craig Leres.
NAME
prifac - prime-factor a list of numbers

SYNOPSIS
prifac @[ ( <int> | <int>-<int> ) ... @]

DESCRIPTION
The program will compute the prime factors of each number you give it as an argument. If no arguments are given, it will read numbers from standard input, one per line. In addition to simple integers, you can also specify a range of integers as "<int>-<int>". Here are some examples:

```
% prifac 223092871 2147483647
223092871 = 317 * 703763
2147483647 = 2147483647
% prifac 65534-65538
65534 = 2 * 7 * 31 * 151
65535 = 3 * 5 * 17 * 257
65536 = 2 ** 16
65537 = 65537
65538 = 2 * 3 ** 2 * 11 * 331
```

AUTHOR
Jef Poskanzer
NAME
ratp2 - pass two of ratfor preprocessor

SYNOPSIS
ratp2 [file] ...

DESCRIPTION
Ratp2 is a filter that outputs FORTRAN code in standard order. It allows for some non-standard features. "Implicit" statements are valid and comment lines may start with either 'c' (or 'C') or 'd' (or 'D'). If filename arguments are not provided or if the file "-" is specified, ratp2 reads from standard input.

SEE ALSO
ratp1(tool), rat4(tool)

AUTHOR
Marylou Orayani

BUGS/DEFICIENCIES
Ratp2 does not work if one of the files given is non-random access, e.g., a terminal.

It also does not recognize text in columns 73-80 as comments.
NAME
rse - Screen-oriented real-time editor.

SYNOPSIS
rse <filename>

DESCRIPTION
RSE is a screen-oriented real-time editor. Screen-oriented means that it makes heavy use of the cursor-control features of your terminal to maintain on the screen an image of the part of your file that you are currently editing. Real-time means that commands are executed as soon as they are typed, and the screen image immediately changes to reflect the new state of the file.

SEE ALSO

Any document on MIT's "EMACS" editor.

AUTHOR
Jef Poskanzer

BUGS/DEFICIENCIES
- File size is currently limited to 32,766 characters. Files larger than this are truncated with no warning. 32,766 characters is about 1300 lines of program (25 chars/line) or 800 lines of document (40 chars/line).

- Search commands are not yet implemented.

- Many other less vital commands are also not implemented.
COMMAND SUMMARY

Control commands (prefixed by `~` or just typed as a control character):

- \texttt{C-@} mark the current position
- \texttt{C-A} go to the beginning of the current line
- \texttt{C-B} move backwards one character
- \texttt{C-C} is a prefix for control-meta commands - see below
- \texttt{C-D} delete one character forward
- \texttt{C-E} go to the end of the current line
- \texttt{C-F} move forward one character
- \texttt{C-G} does nothing (i.e., cancel command)
- \texttt{C-H} move backwards one character
- \texttt{C-I} (tab) insert a tab
- \texttt{C-J} (linefeed) insert a newline and a tab
- \texttt{C-K} delete to the end of the current line
- \texttt{C-L} clear and re-display the screen
- \texttt{C-M} (return) insert a newline
- \texttt{C-N} go to the next line
- \texttt{C-O} insert a newline after point
- \texttt{C-P} go to the previous line
- \texttt{C-Q} quote and insert the next character
- \texttt{C-R} reverse incremental search (terminate with `\texttt{~C}`)
- \texttt{C-S} incremental search (terminate with `\texttt{~C}`)
- \texttt{C-T} exchange the characters before and after point
- \texttt{C-U n} repeat next command \textit{n} times (4 if \texttt{`n`} not given)
- \texttt{C-V} move to the next screenfull of text
- \texttt{C-W} delete the characters between point and mark
- \texttt{C-X} is a prefix character - see below
- \texttt{C-Z} is a prefix for control-meta commands - see below
- \texttt{C-[ (escape)} is a prefix for meta commands - see below
- \texttt{C-] is a prefix for meta commands - see below
- \texttt{C-} exit editor with error status
- \texttt{C-<} is a prefix for control commands
- \texttt{C-space} mark the current position
- \texttt{C-(} insert () around the point
- \texttt{C-)} move to matching right parentheses
- \texttt{rubout} delete one character backward

Meta commands (prefixed by `\texttt{~[ (escape) or ~):}`

- \texttt{M-<} go to the beginning of the buffer
- \texttt{M->} go to the end of the buffer
- \texttt{M-B} go to beginning of word
M-D delete to end of word
M-F go to end of word
M-H select next word
M-V move to the previous screenful of text
M-rubout delete to beginning of word

Control-Meta commands (prefixed by ~C or ~Z, or one of the meta prefixes if the character itself is controlified):

C-M-C exit from the editor
C-M-G no action (i.e., cancel command)
C-M-Z exit from the editor

Control-X commands:

C-X ^ expand window
C-X A append selected region to buffer
C-X B change buffers (creating if necessary)
C-X I insert buffer at point
C-X K kill buffer.
C-X P prepend selected region to buffer
C-X S spawn to a shell
C-X C-C prefix for control-x control-meta commands
C-X C-G no action (i.e., cancel command)
C-X C-I insert file at point
C-X C-S save the buffer on the current file
C-X C-X exchange point and mark
C-X C-V discard current buffer and edit file
C-X C-W write file
C-X C-Z prefix for control-x control-meta commands
C-X C-[ prefix for control-x meta commands
C-X C- prefix for control-x meta commands
C-X C- prefix for control-x control commands
C-X M-B go to window to left
C-X M-F go to window to right
C-X M-N go to window below
C-X M-P go to window above
C-X M-G toggle auto fill
C-X M-J toggle auto justify (only justifies in auto fill mode)
C-X C-M-B make window to left
C-X C-M-D delete window
C-X C-M-F make window to right
C-X C-M-G no action (i.e., cancel command)
C-X C-M-N make window below
C-X C-M-P make window above
If you have a keypad, several other commands will work. Keypad keys can also be made control, meta, and so on. In the following, KO-K9 are keys 0-9 on the keypad, F0-F9 are the function keys numbered 0-9, KU, KD, KL, KR are up, down, left, right keys, respectively. 'K.', 'K,' 'K-' are period, commad, and minus on the keypad, 'KE' is the enter key. The prefix keys can be typed in any order.

F1  prefix for meta commands
F2  prefix for control commands
F3  prefix for control-x commands
K   mark the current position
KU  move to previous line
KD  move to next line
KL  move backward one character
KR  move forward one character
M-KL move to beginning of word
M-KR move to end of word
C-X M-KU move to window above
C-X M-KD move to window below
C-X M-KL move to window to left
C-X M-KR move to window to right
NAME
  sepfor - split programs into multiple files

SYNOPSIS
  sepfor [-vd] file ...

DESCRIPTION
  Sepfor is useful for cracking large Fortran and Ratfor programs into
  separate files. Each subroutine or function is placed in a file of the same
  name. Unnamed modules are put into "main01", "main02", etc.

  The "-v" flag causes sepfor to report to STDOUT the name of each routine as
  it is processed.

  The "-d" flag gives sepfor permission to clobber existing files. By default, it
  will not overwrite existing files.

AUTHOR
  Craig Leres

BUGS/DEFICIENCIES
NAME

talko - multi-user real-time screen-oriented write command

SYNOPSIS

talko @[-c@c]@[user ...@]

DESCRIPTION

Talko is used for the same basic purpose as the write command - to communicate between logged-in users. However, talko is much nicer. Some of the advantages are:

- Write only lets two users talk to each other. Talko allows up to 4.

- Write sends its output a line at a time, forcing the users to sit idle for annoyingly long periods. Talko sends each and every character as it is typed.

- Write obeys the usual teletype paradigm for I/O - output a mess of characters followed by a carriage-return line-feed, and hope the users can sort whose characters are whose. Talko is screen oriented - each user has a separate window to type in, and thus all four users can type at the same time with a minimum of confusion.

The more esoteric commands are documented within the program. All you need to know to get started is that what you type, the other users see, and to exit you type a ^Z.

The -c flag on the command line is useful if you are on a slow terminal; it disables the real-time clock.

SEE ALSO

write(tool)

AUTHOR

Jef Poskanzer

BUGS/DEFICIENCIES

Countably infinite.
NAME
teachrse - tutorial for the RSE editor

SYNOPSIS
teachrse

DESCRIPTION

Teachrse is a short course in the RSE screen editor. It is self-explanatory.

SEE ALSO
rse(tool)

FILES
/misc/lib/teachrse

DIAGNOSTICS

- "scinit: can't open termcap file +LIB/termcap" This means that RSE can't locate a termcap file in your lib search path. Since termcap is currently installed in /misc, the usual fix for this message is to add the directory "'/misc/lib" to your LIB environment variable.

AUTHOR
Jef Poskanzer

BUGS/DEFICIENCIES

- There are still some places where teachrse and rse disagree. Rse is wrong.
NAME
tex - run TeX on file

SYNOPSIS
tex [-k] [-q] [-macpkg] file

DESCRIPTION
This tool is an interface to the TeX document ‘typesetting’ program. It will
build an appropriate dcl command for TeX, run TeX and optionally spool the
TeX output to the VMS TeX spooler.

If the -k flag is supplied, the TeX output is "kept" (i.e., not spooled); if the -q
flag is supplied, the TeX output is spooled to the QMS printer; otherwise, it is
spooled to the Versatec.

The optional -macpkg argument is the name of a macro package (e.g. latex)
to be loaded by the TeX typesetting program.

FILES
file.dvi and file.lis are the output and log files created by TeX. These will be in
the working directory if the -k option is used.

BUGS/DEFICIENCIES
The Versatec and QMS spoolers do not send notification to the user telling
whether the file has been printed successfully.

Use of the QMS printer is limited by a quota system. If you need to use the
QMS printer, you must first see the System Manager to be added to the quota
list.

AUTHOR
Originally written by Van Jacobson, modified for TeX82 and the QMS by Jane
Colman.
NAME
tset - set terminal type

SYNOPSIS
tset [type]

DESCRIPTION
Tset tells the system what type of terminal you are on. Acceptable terminal types are given in the termcap file. If the argument to tset is of the form '?name', tset prompts the user with

    Terminal type: (name)

If the user types a carriage return, the terminal type is set to name. Otherwise, the terminal type is set to the name that the user enters.

If no argument is given, tset reinitializes the terminal, and tells the user what type of terminal it thinks he is on.

FILES
~Lib/termcap - the termcap file

SEE ALSO
sclb(lib)

AUTHOR
Todd Hammond
NAME
upd - repeat a command and display the results

SYNOPSIS
upd [-<N>] command [arg ...]

DESCRIPTION
Upd repeatedly executes a command and displays the results on your screen. By default, upd waits 5 seconds between updates. This can be changed with the -<N> flag, where <N> is the time to wait in seconds. The required argument is the name of a tool. Other arguments are used as arguments for the tool. The rest of the arguments are the command to execute.

SEE ALSO
cdlb(lib)

AUTHOR
Craig Leres and Jef Poskanzer
NAME

termcap - terminal capability database

SYNOPSIS

See below.

DESCRIPTION

**Termcap** is a database describing terminals. It is used by the `sclb(lib)` library, and the `tset` tool. Terminals are described in **termcap** by giving a set of capabilities which they have, and by describing how operations are performed.

Entries in **termcap** consist of a number of ":" separated fields. The first field for each terminal is a unique integer to identify it internally. The second field gives the names which are known for the terminal, separated by ":" characters.

Capabilities:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>att</td>
<td>att</td>
<td>set terminal attributes</td>
</tr>
<tr>
<td>autonl</td>
<td>bool</td>
<td>terminal.auto-newlines after last column</td>
</tr>
<tr>
<td>beep</td>
<td>str</td>
<td>ring bell</td>
</tr>
<tr>
<td>bol</td>
<td>str</td>
<td>move to beginning of line - defaults to (-M)</td>
</tr>
<tr>
<td>bs</td>
<td>bool</td>
<td>shorthand for left=(-H)</td>
</tr>
<tr>
<td>cel</td>
<td>str</td>
<td>clear to end of line</td>
</tr>
<tr>
<td>chscroll</td>
<td>str</td>
<td>change scrolling region</td>
</tr>
<tr>
<td>clear</td>
<td>str</td>
<td>clear screen and home</td>
</tr>
<tr>
<td>cols</td>
<td>num</td>
<td>number of columns</td>
</tr>
<tr>
<td>dlc</td>
<td>str</td>
<td>delete character</td>
</tr>
<tr>
<td>dll</td>
<td>str</td>
<td>delete line</td>
</tr>
<tr>
<td>down</td>
<td>str</td>
<td>move down one line</td>
</tr>
<tr>
<td>f0-f9</td>
<td>str</td>
<td>sequence sent by keypad function keys</td>
</tr>
<tr>
<td>hardcopy</td>
<td>bool</td>
<td>this is a hardcopy terminal</td>
</tr>
<tr>
<td>home</td>
<td>str</td>
<td>home cursor</td>
</tr>
<tr>
<td>iim</td>
<td>str</td>
<td>enter insert mode</td>
</tr>
<tr>
<td>ikey</td>
<td>str</td>
<td>enter keypad mode</td>
</tr>
<tr>
<td>inc</td>
<td>str</td>
<td>insert character</td>
</tr>
<tr>
<td>init</td>
<td>str</td>
<td>initialization string</td>
</tr>
<tr>
<td>inl</td>
<td>str</td>
<td>insert line</td>
</tr>
<tr>
<td>k0-k9</td>
<td>str</td>
<td>sequence sent by keypad number keys</td>
</tr>
<tr>
<td>kcomma</td>
<td>str</td>
<td>sequence sent by comma on keypad</td>
</tr>
<tr>
<td>kdown</td>
<td>str</td>
<td>sequence sent by cursor down key</td>
</tr>
<tr>
<td>kenter</td>
<td>str</td>
<td>sequence sent by enter key on keypad</td>
</tr>
<tr>
<td>kleft</td>
<td>str</td>
<td>sequence sent by cursor left key</td>
</tr>
<tr>
<td>kminus</td>
<td>str</td>
<td>sequence sent by minus key on keypad</td>
</tr>
<tr>
<td>kperiod</td>
<td>str</td>
<td>sequence sent by period on keypad</td>
</tr>
</tbody>
</table>
kright str sequence sent by cursor right key
kup str sequence sent by cursor up key
left str move left one column
lins num number of lines
lseg[r][u][l][d] astr sequence to do segment of line
lshigh bool the line segments above are high resolution
move str direct cursor addressing (see below)
noautonl bool terminal does not auto-newline after last column
notilde bool tilde is invisible (Hazeltines)
oim str exit insert mode
okey str exit keypad mode
overstrike bool terminal overstrikes
padc char pad character (default is delete or null)
peep real ms padding for beep
pbol real ms padding for go to beginning of line
pcel real ms padding for clear to end of line
pclear real ms padding for clear screen and home
pdlc real ms padding for delete character
pdll real ms padding for delete to end of line
pdown real ms padding for cursor down
pillow astr sequence to print filled out character
pinc real ms padding for insert character
psdown real ms padding for scroll down
psup real ms padding for scroll up
ptab real ms padding for tab
right str move right one column
sdown str scroll down
sup str scroll up	
		tab str tab to next tab stop (in multiples of 8)
up str move up one line
visdel bool delete is visible (Ann Arbor)
wrttop bool a write in last line last column wraps home
xonoff bool generates xon/xoff if buffer full (tset)

'Att' is a command which describes a sequence which changes the current
character attributes. The format of 'att' is:

    att=action=sequence

where 'action' is of the form:

    ([+-]h|u|r|b|g|a)*)

The characters h, u, r, b, g, and a have the following meanings

    h  highlight (or lowlight if necessary)
    u  underline
    r  reverse video
b  blink
  g  graphics mode
  a  alternate character set

'+' means that the following attribute is turned on, '-' means that the follow­
ing attribute is turned off. If neither '+' nor '-' is specified, '+' is assumed.
'Sequence' is the command sequence that performs the action 'action'. Any
attributes that are not listed are assumed unchanged.

Things of the type 'astr' expect a list of attributes in parentheses, followed
by the sequence to use if those attributes are set. For example, the entry to
do the 'pillow' character on the vt100 might be:

:pillow=(g):

which means that the pillow character is made by sending a space when the
graphics attribute is set.

The entries of the form 'lseg[r][u][l][d]' allow you to draw lines. The 'r', 'u',
'l', and 'd' tell whether the sequence puts a line segment in the positive x­
axis, positive y-axis, negative x-axis, and negative y-axis respectively. (The
directions are given counterclockwise.) Diagramatically:

   u
  l + r
  d

The sequence

:lsegru(l(g)v:

means that to draw a character with the right upper and left parts filled in,
you set the graphics attribute, and send the character 'v'. The character will
be much like an inverted 'T':

\[\begin{array}{c}
| \\
\hline
\end{array}\]

Default characters are '-', '|', and '+'.

For 'chscroll', sub-sequences beginning with % are interpreted specially. '%fc'
and '%lc' send the first and last lines of the scrolling region as a character (1
generates '-A'), respectively. Similarly, '%fn' and '%ln' send the first and last
lines of the scrolling region as a decimal number. '%%' sends the character
'%. (The form '%f+31c' would send the first line of the scrolling region as a
character after adding 31 to it, and similarly for '%'.)

The move attribute is also interpreted specially. Sub-strings of the form
[1c][01234] are looked for. The digit determines what algorithm to use, and
the 'l' or 'c' specifies whether to apply it to the line number or the column
number. The digits defined so far are:
0  Ascii (a value of 1 generates the character 'A')
1  Ascii plus blank (a value of 1 generates the character ' ')
2  Ansi (a value of 23 generates the string "23")
3  Ann Arbor weirdness (sort of BCD; different for 1 and c)
4  Ansi minus 1 (a value of 23 generates the string "22")

Here is a sample termcap entry (it is not necessarily up to date):

20:h19i19iHeathkit H19:lins#24:cols#80:@
  :beep=-G:autonl:clear=@EH:move=@EYl1c1:cel=@EK:@
  :bs:right=@EC:up=@EA:down=-J:sup=-J:sdowN=@EI:@
  :in1=@EL:d1l=@EM:iim=@E@@:oim=@EO:dlc=@EN:@
  :att=g=@EG:att=g=@EF:att=r=@Eq:att=r=@Ep:@
  :pillow=(g):padc=@000:@
  :lshigh:lsegu=(g)’:lsegrl=(g)a:lsegru=(g)b:@
  :lsegu=(g)’:lsegd=(g)’:lsegr=(g)a:lsegl=(g)a:@
  :lsegd=(g)c:lsegul=(g)d:lsegru=(g)e:lsegrd=(g)f:@
  :lsegrd=(g)s:lsegu=(g)t:lsegrul=(g)u:lsegrud=(g)v:@
  :f1=®ES:f2=®ET:f3=®EU:f4=®EV:f5=®EW:@
  :ikey=@E=:okey=@E>:k0=®E?q:k1=®E?q:k2=®E?q:r:@
  :kup=®EA:kdown=®EB:kright=®EC:kleft=®ED:pclear#14:

FILES
+LIB/termcap - file containing terminal descriptions

SEE ALSO
  sclb(lib), tset(tool) UNIX manual entries curses(3X), termcap(5)

AUTHOR
  Jef Poskanzer and Todd Hammond

BUGS/DEFICIENCIES
  'chscroll' and 'move' should use the same type of format for their arguments.
NAME
cdlb - Craig's display library

SYNOPSIS
include "cddef"
call cdinit call cdend call cdredo
integer minlin, mincol, maxlin, maxcol, lin, numlins call cdclear call cdclra (minlin, mincol, maxlin, maxcol) call cdclrl (lin, numlins)
character ch, str() integer lin, col, int, minwid real ri call cdput (ch, lin, col) call cdputs (str, lin, col) call cdputc (str, lin) call cdputn (int, minwid, lin, col) call cdputr (ri, minwid, lin, col)
character cdgfont, cdsngfont, cdnulfont character cdandfont, cdorfont, cdnotfont character f, fl, f2 integer aU call cdfont (f) f = cdgfont (0) f = cdsngfont (att) f = cdnulfont (0) f = cdandfont (f1, f2) f = cdorfont (f1, f2) f = cdnotfont (f1)
integer lin1, col1, lin2, col2 call cdline (lin1, col1, lin2, col2) call cdbox (lin1, col1, lin2, col2)
integer lin, col call cdmove (lin, col)
logical stoponinput call cdplay (stoponinput)
character pmt(), str(), terms(), termch integer status, cdgets, maxlen, lin, col, cdgetn, num status = cdgets (pmt, lin, col, str, maxlen) termch = cdgetx (pmt, lin, col, terms, str, maxlen) termch = cdgetp (pmt, lin, col, terms, str, maxlen) status = cdgetn (pmt, lin, col, num)

DESCRIPTION
Cdlb is a package of routines to help you write screen-oriented (non-cpu-bound) programs. Cdlb keeps track of things in two buffers; one is called the actual image and the other is called the desired image. The actual image always is the same as what is actually on the terminals screen. The actual image is modified to look like the desired image when cdplay() is called.

Normally, you manipulate the desired image and then call cdplay() to draw the screen.

Cdlb uses sclb(lib) to find out about terminal dependent features; it also uses iolb(lib) to perform "raw" terminal i/o. Terminals that use cdlb programs are required to:

(1) not be hardcopy terminals
(2) have the ability to do absolute cursor addressing OR have the ability to do relative cursor addressing and move the cursor to home

(3) have the ability to clear the screen

1. Descriptions of the routines.

cdinit initializes cdlb. Cdinit() clears the actual and desired images and must be the first routine called from cdlb.

cdend terminates cdlb. This routine must be called before exiting.

cdredo force a complete screen refresh. This routine clears both the terminals display and the actual image.

cdclear clears the desired image.

cdclra clears a rectangular area of the desired image. "minlin" and "maxlin" are the first and last lines of the area to clear and "mincol" and "maxcol" are the first and last columns of the area to clear.

cdclrl clears one or more lines of the desired screen image. "lin" is the first line to clear and "numlins" is the number of lines to clear.

cdput puts a character into the desired image. "Ch" is the character to put on the screen and "lin" and "col" are the line and column to put it on.

cdputs puts a string into the desired image. "Str" is the string to put on the screen and "lin" and "col" specify the line and column to place the string.

cdputc puts a string into the desired image, centered. "Str" is the string to put on the desired image and "lin" is the line to put it on.

cdputn puts a number into the desired image. "Int" is the number and "lin" and "col" specify the position. "Minwid" specifies the minimum width and the number is padded with blanks, if necessary.

cdputr puts a real number into the desired image. This routine is otherwise similar to cdputn().

cdfont changes to the desired font. Each character in the desired image has a font. Changing the desired font changes the font that new characters placed in the desired image will receive.

cdgfont returns the description of the current font.

cdsngfont returns the description of a font which has just the attribute "att" set. Possible attributes are ATTRIBUTE_HIGHLIGHT,
ATTRIBUTE_UNDERLINE, ATTRIBUTE_REVERSE, ATTRIBUTE_BLINK, ATTRIBUTE_GRAPHICS, and ATTRIBUTE_ALTERNATE.

cdnulfont returns the description of a font that has no attributes set. This is called the normal font and it that is in effect when cdinit() is called.

cdandfont returns the description of a font that has all attributes that are set in both fonts "f1" and "f2" at once.

cdorfont returns the description of a font that has all attributes that are set in either font "f1" or "f2".

cdnotfont returns the description of a font that has all the attributes that are not in the font "f1".

cdline draws a line. The line is drawn from line "lin1" column "col1" to line "lin2" column "col2". The line must be either horizontal or vertical.

cdmvove specifies the desired cursor position.

cdbox draws a box defined by the two points.

cdplay optimally updates the actual image to look like the desired image.

An important feature to note is that, if "stoponinput" is true., the routine checks every so often whether any new characters have been typed by the user. If a new character is available, the routine will stop updating the screen and return immediately to allow the main program to process the character.

cdgets reads a line via the screen. Cdgets() writes the prompt "pmt" at the position specified by "lin" and "col" and then reads a line from the terminal, which is returned in str. A maximum of "maxlen" characters will be put into "str", although more may be read.

You can erase characters with either delete or backspace. You can erase the line with either control-x or control-u. You can erase the last word with control-w. You can redisplay the screen with control-l or redisplay the line with control-r.

If no string is input, ERR is returned as the status; otherwise OK.

cdgetx reads a line via the screen, specifying terminating characters. Characters are read up to one of the terminator characters in the string "terms".

Characters are read up to one of the terminator characters and the terminating character is returned as the function value.
Otherwise, this is routine is similar to cdgets().

**cdgetp**  Similar to cdgetx(), this routine uses the characters in "str" as the default input. The user can edit the string using delete, ~W, etc., as with cdgetx() and cdgets().

**cdgetn**  read a number via the screen. Cdgetn() prompts for a number as in cdgets() and returns it in num. Cdgets is used. If no number is input, ERR is returned as the status; otherwise OK.

2. Helpful hints.

When first learning how to use cdlb, keep it simple. I.e. don’t fool around with fonts or lines.

Sometimes, the screen will get screwed up by a terminal broadcast or something. Here's one easy way to redraw the screen:

```plaintext
    call cdredo       # force a refresh
    call cdplay (.false.)  # draw the screen again
```

There are really two ways to use cdclear() when you want to put a different image up on the screen. For example:

```plaintext
    call cdinit
    <modify the desired image>
    call cdplay (.false.)
    call cdclear
    <modify the desired image again>
    call cdplay (.false.)
```

In this example, the first time cdplay() is called, the screen is updated as expected. The second time cdplay() is called, the screen is updated by changing what is already on the screen.

Now another example:

```plaintext
    call cdinit
    <modify the desired image>
    call cdplay (.false.)
    call cdclear
    call cdredo       # force a redraw
    <modify the desired image again>
    call cdplay (.false.)
```

This example is just like the previous one except the second time cdplay() is called, the screen is cleared and the new characters are simply displayed on the clean screen.
FILES
incl/cdlb - user visible symbols

SEE ALSO
iolb(lib), sclb(lib), termcap(file)

AUTHOR
Craig Leres

BUGS/DEFICIENCIES
The routine names and interfaces should look more like the Unix "curses" package.

Character attributes are handled in an strange way.

See iolb(lib) and sclb(lib) for more bug lists.
NAME

deslb - portable Data Encryption Standard library

SYNOPSIS

include "deslbdef"
integer inf, outf, n, count, f

DESBLOCK(key)
DESKS(ks)

DESBLOCK(block)
character keystr()

Bit bits64(64), bits48(48), bits32(32)
integer sixes(11), fours(8)

character chars(8)

Bit from(), to(), a(), b()
character prefix()

call desMakeKey( keystr, key )
call desEncrypt( inf, outf, key )
call desDecrypt( inf, outf, key )
call desKSPerm( key, ks )
call desEE( ks, block )
call desDE( ks, block )

call des64S( bits64, sixes )
call desS64( sixes, bits64 )
call des64C( bits64, chars )
call desC64( chars, bits64 )
call des48S( bits48, sixes )
call desN32( fours, bits32 )

call desCopyBits( from, to, n )
call desXorBits( a, b, n )
call desZeroBits( a, n )
call desRotateBits( a, n, count )
call desPrintBits( f, prefix, a, n )

DESCRIPTION

"deslb" is a portable implementation of the National Bureau of Standards' Data Encryption Standard. It is a fairly slow implementation: 65 msec per 64-bit block, or 8 msec per character. Thus, encrypting a 5000-character message would take 40 seconds.

The best way to figure out how to use "deslb" is to look at "des". However, if you really want to know what the routines do, here are some descriptions:

desMakeKey turn a key string into a key. The key string is converted into a 64-bit DES key by taking each character and XORing the low-order
7 bits into successive 7-bit slots in the key, treated circularly. Thus, for the 10th character of the string, bit 1 is dropped, bit 2 is XORed into bit 64 of the key, and bits 3-8 are XORed into bits 1-6 of the key. After all the characters have been processed, bits 8, 16, ..., 64 are clobbered so that each byte has odd parity. This imposition of odd parity is required by the standard.

desEncrypt encrypt a file. desEncrypt reads characters from an input file and writes encrypted characters on an output file. It does cipher-block chaining, in which each 64-bit block of cleartext is first XORed with the ciphertext of the preceding block, before being passed to the DES encryption. For the first block, since there is no preceding block, the key is used. On output, each 64-bit block gets broken up into 10 6-bit chunks plus a 4-bit nybble. desEncrypt then adds 32 to each chunk and writes it out as a character. Furthermore, each block of 64 bits \( \Rightarrow \) 11 characters is terminated with a newline. This output processing makes the ciphertext mailable. The price, of course, is that the ciphertext is 1.5 times larger than the cleartext.

desDecrypt decrypt a file. The input to desDecrypt should be a ciphertext produced by desEncrypt, as described above.

desKSPerm turn a 64-Bit key into 16 48-bit internal keys, as required by desEB and desDB.

desEB encrypt a block.

desDB decrypt a block.

In addition to those main routines, there are a number of routines you can use to transform data into and out of the "Bit" format used by deslb. In all of these routines, the bits are numbered as follows: bit 1 is the high-order bit of the byte; bit 8 is the low-order but of the first byte; bit 9 is the high-order bit of the second byte; etc., and similarly for nybbles, sixes, etc.

des64S squash 64 Bits into 10 sixes plus a nybble.

des64S spread 10 sixes plus a nybble into 64 Bits.

des64C squash 64 Bits into 8 8-bit characters.

desC64 spread 8 characters into 64 Bits.

des48S squash 48 Bits into 6 sixes.

desN32 spread 8 nybbles into 32 Bits.
desC7    spread a single character into 7 Bits.

Finally, there are several utility routines used to manipulate "Bit" arrays.

desCopyBits copy an array of Bits.

desXorBits XOR two arrays of Bits.

desZeroBits zero an array of Bits.

desRotateBits rotate an n-bit block left by count bits.

desPrintBits print an array of Bits onto a file.

SEE ALSO
des(tool), crypt(tool), cpylrib(lib)

AUTHOR
Jef Poskanzer
NAME

gamlb - game restriction library

SYNOPSIS

call gaminit ( truename )

Logical fdial, fprio, fcron integer despri Character badlist, truename

call gamlinit ( fdial, fprio, fcron, despri, badlist, truename )

call gamend

DESCRIPTION

This library provides a simple method of restricting game-playing. The current restrictions are:

(1) GUEST may not play games.
(2) No games while logged in on a dial-up line.
(3) No games except during the times listed in the game cron file.
(4) Games are played with a base priority of 2.

In addition, there is an optional security feature to discourage game players from making copies of a game. If the truename parameter in the call to gaminit is non-empty, it is assumed to contain the full pathname that the game is supposed to be run from. Gaminit finds out what pathname it was actually run from. If the two don't match, the copy gets removed, and the user is berated.

Valid times to play are listed in the game cron file. If the current time and date matches at least one entry in this file, then it may well be a "good" time to play a game.

A low level access to the routines is provided thru gamlinit(). If the logical fdial is .true., then dialup use is permitted. If the logical fprio is .true., then priority alteration occurs. If the logical fcron is .true., then the game cron file is checked. If fprio is .true., despri is used to determine the new priority. If despri is positive, then it specifies an absolute priority. If it is negative, then it specifies a delta priority. If it is zero, the default lowered priority, two less than the current priority, is used. Badlist is a string that contains account names that are not allowed to play, seperated by COLONs. Truename behaves as above.
FILES
/misc/lib/gamcron - list of "legal" times to play.

SEE ALSO
enlb(lib)

AUTHOR
Jef Poskanzer. The game cron file and other features added by Craig Leres.

BUGS/DEFICIENCIES

Running at a base priority of two can be dangerous. (See kill(tool) for information on how to un wedge stuck game processes.)

A terminal is ASSUMED to be a dialup if it is has the attributes remote and modem.

Since enlb is kept in iolb, it is necessary to link with iolb. For example:

% rc src/program gamlb iolb
NAME
iolb - raw-mode I/O library

SYNOPSIS

character ch integer seconds logical gotone, avail, ioquick, iotimed, iochav, iopeek call ioinit call ioend ch = iogchar ( ch ) call iopchar ( ch ) gotone = iogquick ( ch ) gotone = iotimed ( ch, seconds ) gotone = iopeek ( ch ) avail = iochav ( 0 )

character str(), buf() logical iogstr integer maxlen, n gotone = iogstr ( str, maxlen ) call iopstr ( str ) call iopbuf ( n, buf ) call ioline ( str )

character buf() integer num, status call ioflush call ioeat call iocrlf call iospaces ( num ) call ioerror ( str, status ) call iofmtstr ( buf )

# VAX ONLY logical oldmode, newmode, ioautobroad, iogbroadcast oldmode = ioautobroad ( newmode ) gotone = iogbroadcast ( buf )

DESCRIPTION
iolb is designed to provide raw-mode I/O to and from the user's terminal. Raw-mode I/O is the simplest thing possible: when the user types a character, it is read by the program; when the program writes a character, it is sent to the terminal. No line-by-line reading, no converting tabs to spaces, spaces to tabs, lower case to upper and upper to lower, no ignoring some characters and intercepting others... nothing but simple read-a-character, write-a-character.

1. Descriptions of the routines.

ioinit initializes iolb. ioinit() must be called before any other routines in iolb.
ioend terminates iolb. This routine must be called before exiting.
iogchar gets a character.
iopchar outputs a character.
iogquick gets a character from the type ahead buffer, if one is available.
iotimed reads a character with timeout. If a character isn't input, in the number of seconds specified .false. is returned. Otherwise, the character read is placed in "ch" and .true. is returned.
iopeek looks to see if a character is in the type ahead buffer. If a character is available, it is placed in "ch" and .true. is returned. If there is not a character in the type ahead buffer, .false. is
returned. Note that the character is not read. This means that
the next call to iogchar() will return the same character iopeek() returned.

iochav
tells if a character is available to be read.

iogstr
attempts to get a line of input. If it is successful, then "str" con-
tains the input characters and .true. is returned. A maximum of
"maxlen" minus one characters will be put into "str", although
more may have been read. The bell is rung when the user
attempts to enter more characters than fit in "str". The returned
string is terminated with an EOS character.

Characters are read up to and including the newline (which is not
returned in "str"). The user can erase characters with either
delete or backspace. Words may be erased with ~W. The entire
input line may be killed with ~U or ~X. The character ~V escapes
the special meaning of any character that follows. Other control
characters that are input are echoed as visible sequences.

If the user attempts to delete more characters than have been
input, .false. is returned.

iopstr
outputs a string of characters terminated with EOS.

iopbuf
outputs "n" characters from "buf".

ioline
writes out characters until the EOS; newlines are translated into
carriage return line feed.

ioflush
flushes the output buffer. This routine is automatically called in
various places, but may be used manually too.

ioeat
flushes the type ahead buffer.

iocrlf
outputs a carriage return and line feed.

iospaces
outputs "num" spaces.

ioerror
terminates an iolb program with a message and status value.

iofmtstr
formats a buffer of characters to contain only printable charac-
ters. Carriage returns, line feeds, and blanks are compressed
into a single blank.

ioautobroad changes iolb's action when a terminal broadcast is received. By
default, broadcast messages are simply echoed verbatim. By cal-
ling this routine with .false., it is possible for user to fetch broad-
cast messages using iogbroadcast() and make them synchronous
with his program. The old setting of broadcast messages is
returned.
iogbroadcast checks to see if a terminal broadcast message is available; if one is available, it is stored in "buf", and .true. is returned. Otherwise, .false. is returned.

SEE ALSO
sclb(lib), cdib(lib)

AUTHOR
Jef Poskanzer and Craig Leres

BUGS/DEFICIENCIES
Yes, there are certainly bugs.
NAME

nsklb - library to stack strings

SYNOPSIS

status = nskinit (stack) status = nskpush (stack, string) status = nskpop (stack, string) status = nskfree (stack)

DESCRIPTION

nskinit creates a stack which the user can store names on, and returns an index meaningful to the stack manipulation routines into 'stack', which is of type 'pointer'. 'Status' is ERR if nskinit is unable to create the stack (probably because of insufficient memory). Otherwise, status is OK.

nskpush adds the string 'string' to the end of the stack 'stack', where 'stack' is the index of the stack returned from the nskinit function. 'Status' is ERR if nskinit is unable to add the string 'string' to the end of the stack. Otherwise, status is OK.

nskpop gets the string which was last pushed on the stack 'stack', and returns it into the string 'string'. 'Stack' is the index of the stack returned from the nskinit function. 'Status' is ERR if the stack is invalid, or if there is nothing on the stack. Otherwise, status is OK.

nskfree removes the stack 'stack' and all the strings in it. 'Stack' is the index of the stack returned from the nskinit function. 'Status' is ERR if the stack is invalid. Otherwise, status is OK.

SEE ALSO

memlb(lib)
NAME
  pvlb - resource locking primitives

SYNOPSIS
  include "pvdef" integer lockword logical truth, pvdolock PVLOCK(lockword)
  PVUNLOCK(lockword) truth = pvdolock ( lockword )

DESCRIPTION
  This library provides primitives useful for locking critical resources in a
  multi-process environment. It assumes some form of memory sharing
  between the processes.

  The basic routine is pvdolock(); it returns .true. if it was able to perform the
  lock, else .false.. Usually, this routine is not called directly. Instead, the macros
  PVLOCK and PVUNLOCK are used. The PVLOCK macro calls pvdolock() repeatedly until it locks the lockword. PVUNLOCK is used to remove a lock.
  By convention, PVUNLOCK should be used to initialize a lockword before the
  first time it is used.

  PVLOCK sleeps for 10 milliseconds between attempts to lock. It should do
  some kind of queueing, but that is harder.

  On the Vax, pvdolock() is implemented as a subroutine that uses the bbssi
  (branch on bit set and set, interlocked) instruction. A lockword is considered
  to be locked if the third bit is set, unlocked if it is not.

SEE ALSO

AUTHOR
  Craig Leres

BUGS/DEFICIENCIES
  Pvlb can't be portable since there isn't a way to do an interlocked test and
  set from high level languages.
NAME
rndlb - random number library

SYNOPSIS
integer i, seed1, seed2, ilow, ihigh, v1, v2, trials
real r, rlow, rhigh, mean, stddev, prob
integer rndint, rndgeo, rndbin, rndpoi
real rnd, rnduni, rndnor, rndexp, rndchi, rndbta, rndF, rndt

call rndini (seed1, seed2)
r = rnd(0)
r = rnduni (rlow, rhigh)
i = rndint (ilow, ihigh)
r = rndnor (mean, stddev)
r = rndexp (mean)
r = rndchi (v1)
r = rndbta (v1, v2)
r = rndF (v1, v2)
r = rndt (v1)
i = rndgeo (prob)
i = rndbin (trials, prob)
i = rndpoi (mean)

DESCRIPTION

All the algorithms in this package are from "The Art of Computer Programming", Vol. 2 (Seminumerical Algorithms), sections 3.2.1 and 3.4.1.

rndini initialize the random number package. This routine must be called before any of the other routines in this package can be called. The two seeds are used to initialize the random numbers in a deterministic manner, so that each time you initialize with the same seeds you will get the same sequence. If you want non-deterministic sequences, use 0 as the seeds, and the current time will be used.

rnd random real number in the range @[0..1). This is the basic random number routine. All of the other routines call this one. It takes an argument, but it is just a dummy. It returns a random real number between 0 and 1.

rnduni random real number in the specified range. This routine takes two arguments, a lower bound and an upper bound. It returns a random real number between the two bounds.

rndint random integer in the specified range. This routine takes two arguments, a lower bound and an upper bound. It returns a random integer between the two bounds, inclusive.
**rndnor** normally distributed random real number. The normal distribution is the well known "bell curve", popular with statisticians. This routine takes two arguments, a mean and a standard deviation. It returns a real number whose probability distribution is the bell curve with the specified mean and standard deviation.

**rndexp** exponentially distributed random real number. This routine returns a real number from the exponential probability distribution with the specified mean.

**rndchi** random real number with the chi-square distribution. This distribution is also known as the gamma distribution of order \( v/2 \). Rndchi could be used to simulate a situation where a set of observed values are compared with their theoretical expectations to see if the differences are significant in a statistical sense. Scintillation counters might be an example.

**rndbta** random real number with the beta distribution. This distribution is less widely used.

**rndF** random real number with the F distribution. This distribution is also known as the variance-ratio distribution. The F distribution arises in the analysis of variance, where one wishes to test whether two populations have identical variances. Consequently, it is important in regression analysis.

**rndt** random real number with the t distribution. You would use the t distribution, instead of the normal distribution, to simulate a case where you are looking at means of small samples, where the standard deviation of the parent population is hard to estimate.

**rndgeo** random integer with the geometric distribution. If some event occurs with a given probability, then the number of independent trials needed until the first event occurs (or between occurrences of the event) has the geometric distribution.

**rndbin** random integer with the binomial distribution. If some event occurs with a given probability \( p \) and if we carry out \( t \) independent trials, then the total number of occurrences \( N \) equals \( n \) with probability:

\[
\binom{n}{t-n} p^{t-n} (1-p)^n
\]

**rndpoi** random integer with the Poisson distribution. The Poisson distribution is related to the exponential distribution as the binomial distribution is related to the geometric: it represents the number of occurrences, per unit time, of an event which can occur at any
instant of time; for example, the number of alpha particles emitted by a radioactive substance in a single second has a Poisson distribution. The probability that $N = n$ ($u$ is the mean) is:

$$-u^n \quad e^{-u} / n!$$

**AUTHOR**

Jef Poskanzer. Parts of the manual entry are due to Ed Thiel. Donald E. Knuth helped a lot.

**BUGS/DEFICIENCIES**

Rndlb is almost, but not quite, portable. As currently written, it requires integers at least 28 bits long. It could be re-written to use 16-bit integers, but you would have to deal with arithmetic overflows. The only parts affected would be rndini, rndseq, and the common block.
NAME
sclb - terminal-independent screen control package

SYNOPSIS

logical  dodumb  call  scinit  call  sclinit (  dodumb )  call  scend
call  scclear  call  schome  call  scbol
integer  n   call  scleft (  n ) call  scright (  n ) call  scup (  n ) call  scdown (  n )
integer  lin,  col  call  scmove ( lin,  col )

logical  truth,  scdisp  character  ch   truth  =  scdisp (  ch )

DESCRIPTION

Sclb  is  a  package  of  routines  which  lets  you  do  cursor  addressing,  screen  clearing,  and  other  screen  functions  in  a  terminal-independent  way.  It  is  basically  a  convenient  interface  to  the  termcap  file.

The  user  must  define  the  environment  variable  TERM  to  a  valid  terminal  type  as  found  in  the  termcap  file  when  using  a  sclb  program.

Descriptions  of  the  routines:

scinit  initialization  for  the  screen  package.  This  routine  must  be  called  before  any  others  in  this  package.

sclinit  low  level  initialization  for  the  screen  package.  This  routine  may  be  called  in  place  of  scinit().  If  "dodumb"  is  .false.,  then  sclb  will  behave  just  as  it  does  when  scinit()  is  called,  i.e.  if  the  termcap  file  cannot  be  opened,  if  the  TERM  variable  is  not  available,  or  if  the  terminal  type  cannot  be  located  in  the  termcap  file,  sclb  will  abort  by  calling  error().  If  "dodumb"  is  .true.,  then  instead  of  aborting,  a  built  in  prototype  dumb  terminal  will  be  used.

scend  termination  for  the  screen  package.  You  must  call  this  routine  before  your  program  exits.

scclear  clear  the  entire  screen.

schome  move  the  cursor  to  the  upper  left  corner  of  the  screen.

scbol   move  the  cursor  to  the  beginning  of  the  current  line.

scleft  move  the  cursor  to  the  left  n  columns.
scright move the cursor to the right n columns.

scup move the cursor up n lines.

scdown move the cursor down n lines.

scmove directly address the cursor.

scdisp returns .true. if the passed character can be displayed, else .false.

FILES
include/sclb - symbol definitions needed if you include sclb.c
include/sclb.c - common block containing user-visible parameters

SEE ALSO
iolib(lib), cdlib(lib), termcap(file),

AUTHOR
Jef Poskanzer
NAME

tdlb - Todd's display library

SYNOPSIS

include tdlb include sclb

call tdinit call tdend call tredo

integer window ( WINDSTRUCTSIZE ), minlin, mincol, maxlin, maxcol call
tdmkwind ( window, minlin, mincol, maxlin, maxcol ) call tdchwind ( window )

integer minlin, mincol, maxlin, maxcol, lin, numlins call tdclear call tdclra ( minlin, mincol, maxlin, maxcol ) call tdclri ( lin, numlins )

call tdmkwind ( window, minlin, mincol, maxlin, maxcol ) call tdchwind ( window )

call tdmkwind ( window, minlin, mincol, maxlin, maxcol ) call tdchwind ( window )

call tdmkwind ( window, minlin, mincol, maxlin, maxcol ) call tdchwind ( window )

call tdmkwind ( window, minlin, mincol, maxlin, maxcol ) call tdchwind ( window )

character ch, str() integer lin, col, int, minwid real rl call tdput ( ch, lin, col )
call tdputs ( str, lin, col ) call tdputc ( str, lin ) call tdputn ( int, minwid, lin, col ) call tdputr ( rl, minwid, lin, col )

integer tdgfont, tdsnfont, tdnulfont integer tdandfont, tdonfont, tdnotfont
integer f, f1, f2, att call tdfont ( f ) f = tdgfont ( 0 ) f = tdsnfont ( att ) f =
tdnulfont ( 0 ) f = tdandfont ( f1, f2 ) f = tdonfont ( f1, f2 ) f = tdnotfont ( f1 )

integer lin1, col1, lin2, col2 call tdbox call tdline ( lin1, col1, lin2, col2 )

integer dl, dc call tdshift ( dl, dc )

integer lin, col call tdmove ( lin, col )

logical stoponinput call tdplay ( stoponinput )

logical tdchav, tdgetk, log integer k character dsget, dspeek, c log = tdchav ( 0 ) c = dsget ( c ) log = tdgetk ( k, c ) c = dspeek ( c )

character prmt(ARB), str(MAXLEN), terms(...) termch integer status, tdgets, maxlen, lin, col, tdgetn, num status = tdgets ( prmt, lin, col, str, maxlen )
termch = dsgetx ( prmt, lin, col, terms, str, maxlen ) status = tdgetn ( prmt, lin, col, num )

DESCRIPTION

Tdlb is a package of routines to help you write screen-oriented programs easily and efficiently. Tdlb keeps track of all the characters currently on the screen in an array called the actual screen image; you, the programmer, tell tdlb what you want the screen to look like by putting characters into the desired screen image. Then, when you have your desired screen all set up, you call tdplay and it optimally converts the actual screen to the desired screen.
Tdlb uses sclb(lib) to control the screen, so it is terminal-independent.

Descriptions of the routines:

_tdinit_ initialize the display manager package. Dsinit clears both the external screen and the internal screen images. It must be called before the display manager package can be used.

_tdend_ terminate the display manager package. This routine must be called before exiting.

_tdredo_ force a complete refresh. This routine clears the screen and forces tdplay to re-display the entire screen image the next time it is called.

_tdmkwind_ specify the coordinates of a window. A window is a region of the screen. If tdchwind is later given these coordinates, tdlb will work relative to them. For example, tdclear will clear just this window. Since the default window is the whole screen, tdclear normally clears the whole screen. minlin and maxlin are the first and last lines of the window, mincol and maxcol are the first and last columns of the window. The description of the window will be put in 'window'.

_tdclear_ clear the current window in the desired image. When tdclear is called, it clears the current window in the desired screen image so that the next time tdplay is called, the window is empty. Dsclear does not clear the window immediately. To clear the entire screen immediately, you should call tdredo. The window is cleared to blanks in the normal font.

_tdclra_ clear a rectangular area of the desired screen image. minlin and maxlin are the first and last lines of the area to clear, mincol and maxcol are the first and last columns of the area to clear.

_tdclrl_ clear one or more complete lines of the desired screen image. lin is the first line to clear, numlins is the number of lines to clear.

_tdpput_ put a character into the desired screen image. ch is the character to put on the screen, lin and col are the line and column to put it on.

_tdputs_ put a string into the desired screen image. This routine puts the string into the screen image starting at the specified position and proceeding to the right. If the string would go past the edge of the screen, it is truncated. str is the string to put on the screen, line
lin and column col is the place to write the string.

`tdputc` put a string into the desired screen image, centered. str is the string to put on the screen, lin is the line to put it on.

`tdputn` put a number into the desired screen image. int is the number to put on the screen, line lin and column col is the place to put the beginning of the number. Spaces are put at the beginning of the number so that it has length at least minwid.

`tdputr` put a real number into the desired screen image. rl is the number to put on the screen, line lin and column col is the place to put the beginning of the number. Spaces are put at the beginning of the number so that it has length at least minwid.

`tdfont` change to the font described. The user can create a description of a font by using `tdsngfont`, `tdnulfont`, `tdandfont`, `tdorfont`, and `tdnotfont`. f is the font to change to.

`tdgfont` return the description of the current font. Typically, you use this routine to change fonts temporarily, restoring the old font with `tdfont`.

`tdsngfont` return the description of a font which has just the attribute 'att' set. Possible attributes are `ATTRIBUTE_HIGHLIGHT`, `ATTRIBUTE_UNDERLINE`, `ATTRIBUTE_REVERSE`, `ATTRIBUTE_BLINK`, `ATTRIBUTE_GRAPHICS`, and `ATTRIBUTE_ALTERNATE`.

`tdnulfont` returns the description of a font that has no attributes set. This is the normal font.

`tdandfont` returns the description of a font that has all attributes that are set in both fonts f1 and f2 at once.

`tdorfont` returns the description of a font that has all attributes that are set in either font f1 or f2.

`tdnotfont` returns the description of a font that has all the attributes that are not in the font f1. This is usually useful in connection with `tdandfont` to remove certain attributes from the description of a font.

`tdbox` draw a box around the current window. If you are going to use `tdbox`, remember to leave enough space between the current window and adjacent windows to draw the box, or else characters in adjacent windows will be overwritten.

`tdline` draw a line. The line is drawn from line lin1 column col1 to line lin2 column col2. The line must be either horizontal or vertical. Tdlb is
not currently able to draw other types of lines.

**tdshift** shifts the current window dl (delta lines) lines down and dc (delta columns) to the right. Thus tdshift is a slight extension of scrolling. Any characters that would be moved past the edges of the window are lost, and blanks (in the current font) are put in any position of the window that doesn't have something moved to it. The normal way to scroll up is to call tdshift with '-1' for delta lines, and '0' for delta columns.

**tdmove** specify the desired cursor position. This routine lets you specify where you want the cursor to end up when tdplay terminates. It does not actually move the cursor - it saves the coordinates, and tdplay does the moving. lin is the desired line, col is the desired column.

**tdplay** optimally output the desired screen image onto the screen. Given the desired screen image, which describes the way you want the screen to look, and the actual screen image which describes the way the screen currently looks, this routine converts the current state into the desired state.

An important feature to note is that, if 'stoponinput' is .true., the routine checks every so often whether any new characters have been typed by the user. If a new character is available, the routine will stop updating the screen and return immediately to allow the main program to process the character. The next time tdplay is called, it picks up where it was interrupted.

**tdchav** returns a true value if and only if the user has typed a character that has not yet been read.

**dsget** read a character from the terminal. If you have not yet typed a character, tdib waits until you type one. 'c' is the character read.

**tdgetk** read a possibly keypad character from the terminal. If the character is a keypad character, tdgetk returns the number of the keypad character in 'k'. Otherwise, tdgetk returns the character in 'c'. Dsgetk returns .true. if and only if the character read was from the keypad. Possible keypad characters are KEYPAD_0 through KEYPAD_9, KEYPAD_F0 through KEYPAD_F9, KEYPAD_UP, KEYPAD_LEFT, KEYPAD_DOWN, KEYPAD_RIGHT, KEYPAD_PERIOD, KEYPAD_COMMA, KEYPAD_MINUS, and KEYPAD_ENTER.

**dspeek** returns the next character you type. If you haven't typed a character, dspeek waits for you to do so. This character is not actually 'read' however: the next call to dsget will return this character as well. 'c' is the character that was typed.
tdgets  read a line via the screen. Dsgets writes the prompt prmt starting
at line lin column col, reads a line from the terminal, and returns
it in str. A maximum of maxlen characters will be put into the
string, although more may be read. Characters are read up to and
including a NEWLINE. You can erase characters with either the
rubout or the backspace keys, erase the last word with ^W, or
erase the whole line with ^X or ^U. ^R will retype the line (this is
mainly useful on hardcopy terminals), and ^L will redraw the
screen. The string will be terminated with EOS, as usual. If no
string is input, ERR is returned as the status; otherwise OK.

dsgetx  read a line via the screen, specifying terminating characters. This
routine is similar to tdgets, except the user specifies a list of charac-
ters (terminated by and EOS) in 'terms' that can terminate
input of the string. The character that actually terminates enter-
ing of the string is returned in termch. The terminating character
is not put in the string returned.

tdgetn  read a number via the screen. Dsgetn prompts for a number as in
tdgets and returns it in num. Dsgets is used. If no number is input,
ERR is returned as the status; otherwise OK.

FILES
incl/tdlb - some user-visible symbols

SEE ALSO
iolb(lib), sclb(lib), termcap(file), UNIX manual entries curses(3X), termcap(5)

AUTHOR
    Jef Poskanzer and Todd Hammond

BUGS/DEFICIENCIES
The names and calling sequences should probably look more like the Bell
Laboratory's new "curses" package. In particular, character attributes are
handled in an unusual way, and windows should be probably be handled
differently. Unfortunately, as of the time this manual entry was written, the
complete specifications for this "curses" are not available. Bell Laboratories
"curses" is different in several ways from Unix "curses".

Absolute cursor addressing does not work on hardcopy terminals. (This
includes home cursor and cursor move.) Tdlb doesn't handle character attrib-
tutes on "magic cookie" terminals (terminals that that require a space on
the line to record a change of character attributes (terminology due to Mark
Horton of Bell Labs)). These are rather esoteric problems.

Tdlb will use special "features" on terminals even if they require huge
amounts of padding and so on. (For example, there are terminals that have
half-a-second insert line delays.) The best solution for this sort of problem is
to remove the special features from the termcap file. Cursor motion is still
optimal (or close too it), however.

Tdlb uses a hash to keep track of changes in the lines, and occasionally the hash will match when the lines are really different. This causes tdlb to not change the line in question. For normal text, this is a very rare problem.
NAME
tslb - library to get and set terminal characteristics

SYNOPSIS
call tsinit status = tsset ( perm ) call tsend

log = tsgautonl ( 0 ) log = tsgbroadcast ( 0 ) cols = tsgcols ( 0 ) ms = tsgcrfill
( 0 ) log = tsgdialup ( 0 ) log = tsgform ( 0 ) log = tsghardcopy ( 0 ) ms =
tsgfill ( 0 ) lins = tsglins ( 0 ) call tsgspeed ( ispeed, ospeed ) log = tsgtabs ( 0 )
type = tsgtype ( 0 )

call tssautonl ( log ) call tssbroadcast ( log ) status = tsscols ( cols ) status =
tsscrfill ( ms ) call tssform ( log ) call tsshardcopy ( log ) status = tssfill ( ms )
status = tsslins ( lins ) status = tssspeed ( ispeed, ospeed ) call tsstabs ( log )
status = tsstype ( type )

DESCRIPTION
These routines get and set the terminal type. Tsinit must be the first thing
called in this library. Tsend must be the last thing called.

If you want to set the terminal characteristics, call the routines starting with
'tss' to specify any of the characteristics you want to change, and then call
tssset to actually set them. The following code would temporarily disable
broadcasts, and later reenable them:

logical tsgbroadcast, oldbroadcast
integer tsset

# initialize tslb
call tsinit

# get old setting of broadcasts
oldbroadcast = tsgbroadcast ( 0 )

# turn broadcasts off
call tssbroadcast ( .false. )
if ( tsset ( TS_TEMPORARY ) != OK )
  call error ( "can't turn off broadcasts" )
.
.

# restore broadcasts to their original setting
call tssbroadcast ( oldbroadcast )
if ( tsset ( TS_TEMPORARY ) != OK )
  call error ( "can't restore setting of terminal broadcasts" )

# end tslb (deallocate the channel to the terminal, etc)
call tsend
tsinit  Initialize tslb. This must be the first thing called in tslb.

tsset  Set terminal characteristics. Call tsset after you have specified the characteristics that you want to change with the routines starting with tss. The argument can be TS_PERMANENT or TS_TEMPORARY which tells tslb whether to set the permanent or temporary characteristics. (Changes to temporary characteristics disappear if the user logs out. Changes to permanent characteristics remain forever. Changes to permanent characteristics, however, do not change the temporary characteristics.) OK is returned if tsset was able to set the characteristics. ERR is returned otherwise.

tsend  End tslb. This must be the last thing called in tslb.

tsgautonl  See if system automatically does a carriage-return linefeed after a character is written on the last column of the line. (Note that this does not apply to terminals in passall mode.)

tsgbroadcast  See if other users should be able to write messages to the terminal.

tsgcols  Get the number of columns per line.

tsgcrfill  Get the carriage return delay needed. The carriage return delay is a real number, given in milliseconds.

tsgdialup  See if terminal is a dialup. (On RTSG's Vax, the operating system sometimes thinks that terminals are dialups when they really aren't.)

tsgform  See if terminal has hardware form feeds. (On a crt, find whether the terminal has clear screen and home cursor.) If so, control-L is the form feed character.

tsghardcopy  See if terminal is a hardcopy terminal.

tsglfill  Get the carriage return delay needed. The line feed delay is a real number, given in milliseconds.

tsglins  Find out how many lines there are per page. (Or, on a crt, how many lines there are on the screen.)

tsgspeed  Get terminal's input and output speeds. Both the input and output speeds are real numbers, given in milliseconds per character. Definitions for the input and output speeds are given in the include file tslb. For example, TS_SPEED_9600 is defined to be 0.9600 characters per second.
tsltab See if terminal has hardware tabs. If so, the tab character is \~l.

tsgtype Gets the terminal type. See termcap(file).

setsauto Set whether the system automatically does a carriage-return linefeed after a character is written on the last column of the line.

tssbroadcast Set whether other users should be able to write messages to the terminal.

tsscols Set the number of columns per line.

setscraf Set the carriage return delay needed. The carriage return delay is a real number, given in milliseconds.

setsform Set whether the terminal has hardware form feeds. (On a crt, set whether the terminal has clear screen and home cursor.) If so, control-L is assumed to be the form feed character.

setshardcopy Set whether the terminal is a hardcopy terminal.

setslfill Set the line feed delay needed. The line feed delay is a real number, given in milliseconds.

setslins Set the number of lines per page. (Or, on a crt, the number of lines per screen.)

setspspeed Set terminal speed. The speeds are given in milliseconds per character. See tspspeed.

setsstabs Set whether the terminal has hardware tabs.

setsstype Set the terminal type to 'type'.

FILES
+INCL/tsltab - all programs needing TS_TEMPORARY or TS_PERMANENT should include this

SEE ALSO
tset(tool), termcap(file), iolb(lib)

AUTHOR
Todd Hammond
BUGS/DEFICIENCIES

It would be nice if the user could get and set characteristics of terminals other than his own. The terminal to use could be an argument to tslb.

The 'tsgremote' primitive is not accurate on the RTSG Vax.