Today's Lecture

1. **grep**, programmer's best friend

2. Regular expressions

3. An inside look at an editor that embodies **grep**
   (signals, setjmp/longjmp)

4. **sed**

5. A mailbot – a **sed**/**ksh** application

grep overview

Uses *patterns* to select lines from files.

Patterns specified via *regular-expressions*.

History: `g/re/p` from `ed` editor.

Syntax: **grep** [options] regular-expression [file...]

grep: Examples

Meta-character dot (`.`) matches any single character.

Meta-character star (`*`) matches any number of occurrences of the patter (RE) immediately to its left.

Words with all vowels in alphabetical order:

```
$ grep "a.*e.*i.*o.*u" /usr/dict/words
adventitious
facetious
sacrilegious
$
```

Weird letter patterns:

```
$ grep hsh /usr/dict/words
dachshund
dachshunde
dachshunds
earthshaking
earthshine
messiahship
roughshod
$
```


grep: More examples

Anagrams of "spear"

```
$ DICT=/usr/dict/words
$ grep ^.....$ $DICT | grep s | grep p | grep e | grep a | grep r
parse
prase
presa
spare
spear
$
```

How about "emanate"?

```
$ grep ^.......$ $DICT | grep e.*e|grep m|grep a.*a|grep n|grep t
emanate
manatee
$
```

Another way, when there's no repeated letter:

```
$ grep '^[triangle]*$' $DICT | grep ........ | grep -v '\(.\).*\1'
altering
integral
triangle
$
```

To decipher the last filter we need to understand *holding spaces*.

grep: Holding spaces, Back-references

The back-reference \n, where n is a single digit, matches the substring previously matched by the nth parenthesized, *i.e.* `\(···\)`, subexpression of the regular expression.

Per line, up to nine (9) such back-references can be stored.

For example,

```
$ grep '\(.....\).*\1' $DICT
auriculoventricular
chincherinchee
chincherinchees
denationalization
internationalization
kinnikinnick
micromicron
nationalization
rationalization
$
```


grep: More examples

All words containing three pairs of the same character:

```
$ grep '.*\(.\)\1.*\1\1.*\1\1.*' /usr/dict/words
expressionlessness
possessedness
possessiveness
prepossessingness
stresslessness
$
```

Five letter palindromes:

```
$ grep '^\(.\)\(.\).\2\1$' /usr/dict/words
alula
civic
deled
kayak
level
ma'am
madam
radar
rotor
solos
tenet
$
```

What about general n letter palindromes?

grep: More palindromes

Regular expressions aren't "powerful" enough to express general palindromes, among other languages.

A `ksh` script to the rescue:

```
#!/bin/ksh
# mk-palindrome-re.ksh

HOLD='\(.\)'        # RE to "remember" a character
FRONT=""
TAIL=""
MIDDLE=""
integer i=$1
integer count=1

if ((i % 2 == 0))
then    MIDDLE=""
else    MIDDLE="."
        i=i-1
fi
while ((i > 0))
do      FRONT="$FRONT$HOLD"
        TAIL="\\$count$TAIL"
        count=count+1
        i=i-2
done
REGEX="^$FRONT$MIDDLE$TAIL\$"
print $REGEX

# end of mk-palindrome-re.ksh
```


grep: More palindromes (cont.)

And, with a bit more glue

```
#!/bin/ksh
# palindrome.ksh
DICT=/usr/dict/words
integer start=$1
integer stop=$2
integer i=$start
while ((i <= stop))
do      print "$i letter palindromes in $DICT:"
        REGEX=$(mk-palindrome-re.ksh $i)
        grep "$REGEX" $DICT
        i=i+1
done
# end of palindrome.ksh
```

we find:

```
$ palindromes.ksh 6 19
6 letter palindromes in /usr/dict/words:
denned
redder
terret
7 letter palindromes in /usr/dict/words:
reviver
rotator
8 letter palindromes in /usr/dict/words:
9 letter palindromes in /usr/dict/words:
10 letter palindromes in /usr/dict/words:
11 letter palindromes in /usr/dict/words:
12 letter palindromes in /usr/dict/words:
13 letter palindromes in /usr/dict/words:
14 letter palindromes in /usr/dict/words:
15 letter palindromes in /usr/dict/words:
16 letter palindromes in /usr/dict/words:
17 letter palindromes in /usr/dict/words:
18 letter palindromes in /usr/dict/words:
19 letter palindromes in /usr/dict/words:
$
```

grep+ksh: One last example

Find the longest line in a file.
Again, regular expressions alone are not powerful enough.

A **ksh** script that uses **grep** does the trick:

```
#!/bin/ksh
# longest.ksh

GREP=grep
SRC=$1
ONE_CHAR="."    # RE to match a single char
START="^"       # RE to match beginning of a line
STOP="\$"       # RE to match end of a line
REGEX=""
while $($GREP -s "$START$REGEX$ONE_CHAR$STOP" $SRC)
do    REGEX=$REGEX$ONE_CHAR
done

$GREP $START$REGEX$STOP $SRC
# end of longest.ksh
```

```
$ longest.ksh /usr/dict/words
electroencephalographic
$
```


grep options

-n print line number on output

-v invert: print lines that don't match

-i ignore case of input letters

-c count lines instead of printing them

-s return only status

-l print names of files that contain matches

Regular expressions

c	a character matches itself, unless it is a metachar (`\ [ ] * + ^ $`)
.	dot matches any character
\\c	the usual escape character; escapes any special meaning c may have
[*char set*]	matches one of the characters in the set. If the first character in the set is ^, it matches a character *not* in the set. The shorthand c_1-c_2 is used to specify a set of characters c_1 upto c_2, inclusive.
*RE**	matches regular expression *RE* zero or more times
RE+	matches regular expression *RE* one or more times

Regular expressions (cont.)

r_1r_2	the concatenation of two regular expressions r_1 and r_2 is a regular expression that first matches r_1 and then r_2
$r_1 \| r_2$	the alternation of two regular expressions r_1 and r_2 is a regular expression that matches either r_1 or r_2
\\(*RE*\\)	places the string matching *RE* in the next available holding space, numbered left to right from one.
\\n	where n is a single digit, matches the string stored in holding space n
\\<	matches the null string at the beginning of a word
\\>	matches the null string at the end of a word
^	matches the beginning of a line
$	matches the end of a line

jed: A Brief Look

`jed` is a line editor
`jed` shares lineage with `ed` and hence with `grep`

lines	words	chars	file	
29	63	495	`main.c`	sets things up
52	136	791	`sf.c`	signals, error handling, exit
177	461	2599	`ef.c`	parsing and addressing
157	431	2758	`commands.c`	command loop
139	395	2311	`fio.c`	text buffers, file I/O
63	142	833	`tio.c`	terminal I/O
61	142	988	`defs.h`	defines
42	147	991	`errmsg.c`	error messages
354	1039	6422	`grep_code.c`	grep, basically
1074	2956	18188		total

command	files			
`a`	`ef.c`	`tio.c`		Append
`d`	`ef.c`			Delete
`E/e`	`ef.c`	`fio.c`		(FORCED) edit
`i`	`ef.c`	`fio.c`	`tio.c`	insert
`\n`	`ef.c`	`fio.c`	`tio.c`	print next line and move
`p`	`ef.c`	`fio.c`	`tio.c`	print
`Q/q`	`ef.c`	`fio.c`		(FORCED) quit
`r`	`ef.c`	`fio.c`		read
`s`	`ef.c`	`grep_code.c`		substitute
`u`	`ef.c`			undo
`W/w`	`ef.c`	`fio.c`		append/write
`=`	`ef.c`			number of lines

setjmp & longjmp

Library functions **setjmp** & **longjmp** provide non-local gotos. They are usually used to recover from errors without having to "unwind" the call stack.

`setjmp` saves the machine state (in a struct called a `jmp_buf`) for later use by `longjmp`.

`longjmp` restores the machine state to one previously captured by `setjmp`

Their implementation is highly machine specific;
details for `phoenix` from `<setjmp.h>`:

```
#ifndef __sparc_setjmp_h
#define __sparc_setjmp_h
/*
 * onsstack,sigmask,sp,pc,npc,psr,g1,o0,wbcnt (sigcontext).
 * All else recovered by under/over(flow) handling.
 */
#define _JBLEN  9
typedef int jmp_buf[_JBLEN];
```


setjmp & longjmp man pages


```
SETJMP(3V)              C LIBRARY FUNCTIONS              SETJMP(3V)

NAME
     setjmp, longjmp - non-local goto

SYNOPSIS
     #include <setjmp.h>

     int setjmp(env)
     jmp_buf env;
     void longjmp(env, val)
     jmp_buf env;
     int val;

DESCRIPTION
     setjmp() and longjmp() are useful for  dealing  with  errors
     and  interrupts  encountered  in a low-level subroutine of a
     program.

     The macro setjmp() saves its stack environment  in  env  for
     later  use  by longjmp().  A normal call to setjmp() returns
     zero.  setjmp() also saves the register environment.   If  a
     longjmp()  call  will  be  made,  the  routine  which called
     setjmp() should not return until  after  the  longjmp()  has
     returned control.
```


setjmp & longjmp man pages (cont.)


```
     longjmp() restores the environment saved by the last call of
     setjmp,  and  then returns in such a way that execution con-
     tinues as if the call of  setjmp()  had  just  returned  the
     value val to the function that invoked setjmp(); however, if
     val were zero, execution would continue as if  the  call  of
     setjmp()  had  returned one.  This ensures that a ``return''
     from setjmp() caused by a call  to  longjmp()  can  be  dis-
     tinguished from a regular return from setjmp().  The calling
     function must not itself have returned in the interim,  oth-
     erwise  longjmp()  will  be  returning control to a possibly
     non-existent environment.  All memory-bound data have values
     as  of the time longjmp() was called.  The CPU and floating-
     point data registers are restored to the values they had  at
     the  time that setjmp() was called.  But, because the regis-
     ter storage  class  is  only  a  hint  to  the  C  compiler,
     variables declared as register variables may not necessarily
     be assigned  to  machine  registers,  so  their  values  are
     unpredictable after a longjmp().  This is especially a prob-
     lem for programmers trying to  write  machine-independent  C
     routines.

     setjmp() and longjmp() save and restore the signal mask (see
     sigsetmask(2)),

     None of these functions save or restore  any  floating-point
     status or control registers,
```

Unix Signals, in a Nutshell

Processes may need to *asynchronously* communicate with one another. (Process A signals process B)

For example, it is often necessary to deliver a `^C` to a process.

Signals also allow recovery from errors (*e.g.* divide-by-zero, segmentation fault).

Program can install *handlers* (C functions) for specific signals. Upon receipt of a handled signal, control is passed to the installed handler, if one exists.

Programming with signals is closely related to programming with interrupts and to concurrent programming.

signal man page


```
SIGNAL(3V)              C LIBRARY FUNCTIONS              SIGNAL(3V)

NAME
     signal - simplified software signal facilities

SYNOPSIS
     #include <signal.h>

     void (*signal(sig, func))()
     void (*func)();

DESCRIPTION
     signal() is a  simplified  interface  to  the  more  general
     sigvec(2)  facility.   Programs that use signal() in prefer-
     ence to sigvec() are more likely to be portable to all  sys-
     tems.

     A signal is generated by some abnormal event, initiated by a
     user  at  a  terminal  (quit, interrupt, stop), by a program
     error (bus error,  etc.),  by  request  of  another  program
     (kill),  or  when  a process is stopped because it wishes to
     access its control terminal while  in  the  background  (see
     termio(4)).  Signals are optionally generated ....
     ...  Most signals cause termination of  the  receiving
     process  if  no  action is taken; some signals instead cause
     the process receiving them to be stopped, or are simply dis-
     carded  if  the process has not requested otherwise.  Except
     for the SIGKILL  and  SIGSTOP  signals,  the  signal()  call
     allows  signals  either  to  be ignored or to interrupt to a
     specified location.  The following is a list of all  signals
     with names as in the include file <signal.h>:
```


signal man page


```
          SIGHUP     1     hangup
          SIGINT     2     interrupt
          SIGQUIT    3*    quit
          SIGILL     4*    illegal instruction
          SIGTRAP    5*    trace trap
          SIGABRT    6*    abort (generated by abort(3) routine)
          SIGEMT     7*    emulator trap
          SIGFPE     8*    arithmetic exception
          SIGKILL    9     kill (cannot be caught, blocked, or ignored)
          SIGBUS     10*   bus error
          SIGSEGV    11*   segmentation violation
          SIGSYS     12*   bad argument to system call
          SIGPIPE    13    write on a pipe or other socket with no one to read it
          SIGALRM    14    alarm clock
          SIGTERM    15    software termination signal
          SIGURG     16@   urgent condition present on socket
          SIGSTOP    17+   stop (cannot be caught, blocked, or ignored)
          SIGTSTP    18+   stop signal generated from keyboard
          SIGCONT    19@   continue after stop
          SIGCHLD    20@   child status has changed
          SIGTTIN    21+   background read attempted from control terminal
          SIGTTOU    22+   background write attempted to control terminal
          SIGIO      23@   I/O is possible on a descriptor (see fcntl(2V))
          SIGXCPU    24    cpu time limit exceeded (see getrlimit(2))
          SIGXFSZ    25    file size limit exceeded (see getrlimit(2))
          SIGVTALRM  26    virtual time alarm (see getitimer(2))
          SIGPROF    27    profiling timer alarm (see getitimer(2))
          SIGWINCH   28@   window changed (see termio(4) and win(4S))
          SIGLOST    29*   resource lost (see lockd(8C))
          SIGUSR1    30    user-defined signal 1
          SIGUSR2    31    user-defined signal 2

     The starred signals in the list above cause a core image  if
     not caught or ignored.
```


signal man page


```
     If func is SIG_DFL, the default action  for  signal  sig  is
     reinstated;  this  default is termination (with a core image
     for starred signals) except for signals marked with @ or  +.
     Signals  marked  with  @  are  discarded  if  the  action is
     SIG_DFL; signals marked with + cause the  process  to  stop.
     If  func  is  SIG_IGN the signal is subsequently ignored and
     pending instances of the signal are  discarded.   Otherwise,
     when the signal occurs further occurrences of the signal are
     automatically blocked and func is called.

     A return from the function unblocks the handled  signal  and
     continues  the  process  at  the  point  it was interrupted.

     The value of signal() is the previous (or initial) value  of
     func for the particular signal.

     After a fork(2V) or vfork(2) the child inherits all signals.
     An  execve(2V)  resets  all  caught  signals  to the default
     action; ignored signals remain ignored.

RETURN VALUES
     signal()  returns  the  previous  action  on  success.    On
     failure, it returns -1 and sets errno to indicate the error.

NOTES
     The handler routine can be declared:
          void handler(sig, code, scp, addr)
          int sig, code;
          struct sigcontext *scp;
          char *addr;
     Here sig is the signal number; code is a parameter  of  cer-
     tain  signals  that  provides  additional  detail;  scp is a
     pointer to the sigcontext structure (defined in <signal.h>),
     used to restore the context from before the signal; and addr
     is additional address information.
```

jed: main.c


```
#define DEFINE_GLOBALS
#include "defs.h"

main( )
{
        init();
        setjmp(savej);
        commands();
        exit(0);
}

init()
{
        extern void onintr(), really_quit(), onhup();
        lasterr = -1;
        fchange = 0 ;
        nlall = 128;
        zero = (int *)malloc(nlall*sizeof(int));
        dot = dol = zero;
/*
        SIGHUP    1    hangup
        SIGINT    2    interrupt
        SIGTERM   15   software termination signal
*/
        signal(SIGINT, onintr);
        signal(SIGHUP, onhup);
/*
        SIG_DFL   (void (*)())0   default handler
        SIG_IGN   (void (*)())1   ignore signal
*/

        if ((int)signal(SIGTERM, SIG_IGN) == 0)
                signal(SIGTERM, really_quit);
}
```


jed: sf.c


```
#include "defs.h"

void onintr() {
        signal(SIGINT, onintr);
        putchr('\n');
        lastc = '\n';
        error(13);
}

void onhup() {
        signal(SIGINT, SIG_IGN);
        signal(SIGHUP, SIG_IGN);
        if (dol > zero) {
                addr1 = zero+1;
                addr2 = dol;
                io = creat("ed.hup", 0666);
                if (io > 0) putfile();
                }
        fchange = 0;
        exit(0);        /* quitting time */
}

error(int errno) {
        register c;
        lasterr = errno; /* refresh memory */
        wrapp = pflag = count = 0;
        lseek(0, (long)0, 2);  /* seek to the end of stdin */
        peekc = lastc;
        if(lastc)
                while ((c = getchr()) != '\n' && c != EOF)   ;
        if (io > 0) { close(io); io = -1; }
        if ( errno != 13 )
                puts(errmsg[errno]); /*No ? for interrupt.*/
        longjmp(savej, -1);
}

void really_quit()
{        if ( (fchange) && (dol != zero) ) error(30); exit(0);   }
```


sed, Stream Editor

Derived from line editor `ed`

Useful for making repetitive edits to a set of files.

Operates on lines from the input stream, but does not modify the input stream; edits must be separately written to a file. Input lines are buffered in a *pattern space*.

Substitution commands based on regular expressions.

Addressing selects lines to which edits are to be applied.

Addressing is also regular

sed: Syntax & Functions

Command-line syntax:

`sed` [`-n`] [`-e` script] [`-f` sfilename] [filename]

Script syntax:

[address[,address]]function[arguments]

Some `sed` functions:

`d`

delete the pattern space

`p`

copy the pattern space to standard output

`s/`*regular expression*`/`*replacement*`/`*flags*

substitutes the *replacement* string for instances of the *regular expression* in the pattern space.
Some possible *flags*: *n*, `g`, `p`

Many other functions for inserting/appending text, explicit file IO, script control-flow, hold-space management, *etc.*

sed: Addresses

An address is either:

n	decimal line number, *e.g.* 1 is the first line
`$`	which denotes the last line
/RE/	is a context address which selects a pattern space matching the regular expression RE.

If no address is specified, every pattern space is selected.

A single address selects each pattern space that matches.

A pair of addresses

address1,*address2*

inclusively selects the range of pattern spaces from the first pattern matching *address1* to the first pattern matching *address2*

sed: Examples

On `phoenix`, the command `ypcat passwd` displays info about the networked-filesystem's users, *e.g.*:

```
$ ypcat passwd | head -5
pjcorcor:*:17570:29:Patrick J. Corcoran,Macmillan Bldg,8-4395,:/u/pjcorcor:/bin/nologin
jarnheim:ljNazHD.hvX5A:16883:29:Juliette Arnheim,Chemistry Library,8-6173,:/u/jarnheim:/bin/csh
silvias:ggJOCdFeseQOA:13632:35:Silvia A. Salvucci,123 1903 Hall,8-8882,:/u/silvias:/usr/princeton/bin/tcsh
rmsaron:dlvucyFgmLwsk:6096:35:Robin M. Saron,Student,NONE,:/u/rmsaron:/bin/csh
isaaca:laRH6iOuYw4Pg:8632:35:Isaac C. Anderson,405 1901 Hall,8-9078,:/u/isaaca:/bin/csh
$
```

Using `sed` as `grep`:

```
$ ypcat passwd | sed -n -e '/cs333/p'
cs333:nopassword:3687:33:CS333,Computer Science,NONE,:/u/cs333:/bin/csh
$
```

Name, location, & phone only:

```
$ ypcat passwd | sed -n -f nlp.sed | head -5
Patrick J. Corcoran,Macmillan Bldg,8-4395
Juliette Arnheim,Chemistry Library,8-6173
Silvia A. Salvucci,123 1903 Hall,8-8882
Robin M. Saron,Student,NONE
Isaac C. Anderson,405 1901 Hall,8-9078
$
$ cat nlp.sed
s/[^:]*:[^:]*:[^:]*:[^:]*://
s/,:.*//
p
$
```


A Mailbot from `ksh` & `sed`

A mailbot is a program that filters incoming mail messages, extracting and processing messages matching some set of criteria.

Mailbots usually automate some service.

We'll construct a mailbot that upon receipt, automatically processes mail messages containing `cc-compile` in the message's `Subject` line.

We'll assume such a message's body contains a valid C program.

After processing (*i.e.* compiling), our bot will reply to the message. The reply will contain `.o` for the submitted C program (in ASCII form to keep mailers happy).

Mailbot: Things We'll Need

Every message sent to our account will be filtered through our script, `mailbot.ksh`, which will need to:

1. check for a `Subject` line containing only `cc-compile`
2. extract a return address
3. extract the message body (C program)
4. compile the program
5. return the result to the sender

Mailbot: required sed scripts

`extract-hdr.sed`: a script to extract a mail message's header:

```
1,/^$/p
```

`extract-return.sed`: a script to extract a return address from the message:

```
/^Return-Path:.*/{
s/^Return-Path:[ ⟨tab⟩]*//
p
q
}
```

`extract-body.sed`: a script to extract the program (message body):

```
/^$/,$p
```

Note: Above scripts all require `-n` *option to* `sed`

mailbot.ksh


```
#!/bin/ksh
# mailbot.ksh

RECEIPT_DATE=`date`
KEYWORD="cc-compile"
TAB='	'
REGEX="^subject:[ $TAB]*$KEYWORD[ $TAB]*$"

MSG_FILE=$$.txt		# a copy of the message
MSG_SRC=$$.c		# the body of the message
MSG_OBJ=$$.o		# the compilation result
ERROR_LOG=$$.log	# any compilation errors

MAIL=Mail		# must have -s subject option
DIR=/u/username/	# must have global write permissions!

cd $DIR
rm -f $MSG_FILE $MSG_SRC $MSG_OBJ $ERROR_LOG

while read -r line # read stdin into $MSG_FILE
do print -r $line
done >$MSG_FILE

if `sed -n -f extract-hdr.sed $MSG_FILE | grep -i -s "$REGEX" >/dev/null`
then # a request has arrived; find return address and ack receipt
	RETURN_TO=`sed -n -f extract-return.sed $MSG_FILE`
	MAIL_SUBJECT="your request of $RECEIPT_DATE"
	$MAIL -s "$MAIL_SUBJECT" $RETURN_TO <<DONE
You should receive a compiled .o file shortly.
DONE

	# extract the program to compile
	sed -n -f extract-body.sed $MSG_FILE >$MSG_SRC

	# compile it and,
	if `cc -c $MSG_SRC 2>$ERROR_LOG`
	then	# if successful, return it as ASCII
		uuencode $MSG_OBJ result.o | \
			$MAIL -s "result of $MAIL_SUBJECT" $RETURN_TO
	else	# otherwise, send the error log along
		$MAIL -s "$MAIL_SUBJECT failed" $RETURN_TO <$ERROR_LOG
	fi
fi

# end of mailbot.ksh
```


Mailbot: Loose Ends

To wire `mailbot.ksh` into the mail delivery system, put the lines

```
\username
|⟨path-to-mailbot⟩/mailbot.ksh
```

in your `.forward` file.

Note: the mailbot's working directory requires write permissions for "others" since the mailbot is not running as you and it needs to create files.

Be careful with mailbots!

Next Lecture

- Read `man` pages for:

  `lex`

  `yacc`

  `awk`