csh.doc.hyper

COLLABORATORS			
	TITLE : csh.doc.hyper		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 26, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

csh.doc.hyper

1.1 C-Shell documentation

```
C-SHELL 5.19       1991-1992 by U. DOMINIK MUELLER         TUTORIAL

@{ " Installation " link Installation }     How to install csh
@{ " Introduction " link Introduction }     For first time csh users
@{ " Features     " link Features     }     What csh can do
@{ " Restrictions " link Restrictions }     What csh can't do
@{ " Startup      " link Startup      }     What csh does when starting up
@{ " Editing      " link Editing      }     How to enter a command line
@{ " Parser       " link Parser       }     What csh does before calling  ↩
   commands
@{ " Execution    " link Execution    }     How csh tries to execute commands
@{ " Commands     " link Commands     }     Csh's built in commands
@{ " Functions    " link Functions    }     Csh's built in functions
@{ " Variables    " link Variables    }     Csh's special meaning variables
@{ " Programming  " link Programming  }     How to write csh scripts
@{ " Classes      " link Classes      }     Csh's unique file class feature
@{ " Keymaps      " link Keymaps      }     How to program the command line  ↩
   editing
```

1.2 installation

```
You  can simply invoke csh from the command line or at the end
of  your  startup-sequence,  because csh is, from the AmigaDOS
standpoint, not a true shell like l:shell-seg.

@{ " Final installation " link final   }
@{ " Supporting files   " link support }
```

1.3 final

```
                For  a  final  setup,  csh should be resident.  This makes
things faster and saves memory.  Csh is 'pure', so if your
archiver  forgot  to  set  the  'p'  bit, do it right now.
```

```
  Then...

  1. Copy csh anywhere on your disk, e.g. dh1:tools
  2. Add to the bottom of your startup-sequence:
    Stack 8000
    Resident dh1:tools/csh ADD
    csh s:firstlogin.sh
  3. In s:firstlogin.sh, put any commands to be called only once,
     like setmap, assign, setclock. This is a shell script, use #
     for comments, and don't use .key type commands. At the end of
     this file, add
    source s:login.sh
  4. In s:login.sh, you put any commands that need to be executed
     on every invocation, like 'alias', 'set' and so on.
  5. Put the following the s:cli-startup and s:shell-startup, so
     csh will be started in any window opened by NewCLI, NewShell
     or from workbench:
    csh s:login.sh

See
                Scripts
                for details on how to write your startup scripts.
```

1.4 support

```
  I  recommend  to  assign CSH:  somewhere and put your docs and
  csh  scripts  there.   You  can  do the assign from inside the
  shell,  it's  not  needed  at  startup.   The file s:.login is
  executed on every startup if it exists.

  For online help inside csh, put csh.doc to CSH:.  Then you can
  use the 'man' command for any desired command.  Press the HELP
  key for a command list.
```

1.5 introduction

```
                C-Shell  is  a  replacement  for the AmigaDOS command line
  interface.   It's  main  intentions are:  Save the user as
  much  typing as possible, give him/her a fast and powerful
  scripting  language, and do whatever makes controlling the
  Amiga more comforable.


                Author

                Availability

                Credits
```

1.6 credits

Arexx is a program by William Hawes.
Cygnus Ed Professional (C) 1988 CygnusSoft Software.
ARP is copyrighted by the authors.

This version of Shell is the successor of:
1.00 Lattice (c) 1986 Matthew Dillon
2.05 Manx(M) versions by Steve Drew
3.00 ARP (A) versions by Carlo Borreo, Cesare Dieni
4.00 ARP 1.3 versions by Carlo Borreo, Cesare Dieni
5.00 Lattice versions by U. Dominik Mueller

Thanks to:

Matt Dillon, Steve Drew, Carlo Borreo and Cesare Dieni for the previous versions of Csh.

Eddy Carroll, Gary Duncan, Randell Jesup, Steve Koren, Tim MacKenzie, Axel Roebel, Mike Schwartz for their code contributions and suggestions.

Michael Beer, Carl Johan Block, Hubert Feyere, Magnus Heldestad, Carsten Heyl, Alex Liu, Declan McArdle, Svante Eriksson, Luke Mewburn, Peter Orbaek, Allard Siemelink, Roddi Walker, Shimon Weissman and the unknown Swedish guy who found the double-LF bug for their bug reports and feedback.

Roy Haverman, Martin Horneffer, Hans-Christian Huerlimann, Daniel Pfulg, Patrizio Rusconi, Christian Schneider and especially Markus Illenseer for the German translation of the doc.

Olivier Berger for the French translation of the doc.

1.7 author

Csh 5.xx was written by

U. Dominik Mueller
Schulhausstrasse 83
CH-6312 Steinhausen
SWITZERLAND

FIDONET : Urban Mueller@2:302/906 (AUGL)
INTERNET: umueller@amiga.physik.unizh.ch
umueller@iiic.ethz.ch

Feel free to send me all kinds of feedback, flames and flattery. I speak English (obviously), German and French. Please check the 'restrictions' chapter before reporting any bugs and add your csh version and a description of your configuration.

1.8 Availability

The support BBS is AUGL (FIDO address 2:302/906). Numbers:
+41 75 8 20 19
+41 75 8 20 18 (all lines USR/USR Dual)
+41 75 2 15 87
Log in with username 'cshell', password='support'. You may file request there as well, the file name is cshxxx.lha, where xxx is the version number of csh.

The support FTP site is amiga.physik.unizh.ch [130.60.80.80]. The file is named /amiga/csh/cshxxx.lha, where xxx is the version number of csh. You may also try ab20.larc.nasa.gov [128.155.23.64], in the directory /incoming/amiga or /amiga/utilities/shells.

You may distribute this program unmodified and for non-profit only. You may not modify this program and redistribute it! Please contact me if you want to make changes, possibly *before* doing them.

1.9 features

Shell provides a convenient AmigaDos alternative command interface. All its commands are built, which is faster and works fine when no disk is in drive. But Amiga csh is not compatible to UNIX csh nor AmigaDOS (although it can execute AmigaDOS scripts).

Major features include:

EDITING
is freely programmable

completion
of abbreviated file names

history
with history search

terminal
mode, works on VT terminals

PARSER
processes your command lines before passing them

variables
& variable handling (embedded variables)

wildcards
('?', '*' and more)

```
                pipes
                between programs

                execution
                tries to execute a file


                PROGRAMMING
                faster and more powerful than other languages

                commands
                that require no system disk

                functions
                for use in scripts and aliases

                aliases
                with arguments

                scripts
                (w/ gotos and labels)


                GOODIES
                various useful stuff

                scrolling
                internal commands can jump scroll

                classes
                of files, actions on classes

                menus
                lets you attach Intuition menus

  Click to see main differences to
                AmigaShell
                and
                UNIX
```

1.10 goodies

```
                 Some goodies in csh that you should look up:

                scrolling
                internal commands can jump scroll

                closegad
                the effect of the closing gadget can be programmed

                classes
                of files, actions on classes

                menus
                lets you attach Intuition menus
```

```
rback
can be replace by appending &

prompt
and titlebar can show all kinds of data
```

1.11 scrolling

Csh allows you to do quick scrolling in large windows. Quick scrolling means that whenever the cursor reaches the bottom of the window, the text jumps up 3 or more lines at once. However, only the following commands support this:

```
    dir, cat, htype, strings, search, truncate, tee
```

You can choose the number of lines to scroll at once by setting the variable _scroll. Unsetting it or setting it to a value <=1 completely disables quick scrolling.

You can also choose the number lines a window must at least have to turn on the quick scrolling by setting the _minrows variable. (Defaults to 34). Quick scrolling is automatically disabled when the command is redirected. By piping any command to cat, you can force it to quick scroll. Example: List | cat

1.12 closegad

Csh now can be terminated using the closing gadget in the AmigaDOS 2.0 shell window. The closing button provides a 'quit' command. You can define

```
  alias quit "Endcli;quit
```

to assert that the CLI window closes when you click the button.

1.13 menus

It is possible to append Intuition menus to your csh console window. Up to six menus with 16 items every can be installed. Menus can only be selected when the prompt is showing.

```
Usage   : menu [-n] [ title item...item ]
Example : menu Shell   JrComm,,j   Rename,"rename ",r  quit
```

If the item is just a string, that string will be in the menu item. When you select it, it will be put into the prompt and executed.

If there is a comma and after that comma a second string,

this will be the command will be inserted at the prompt. This time you have to add the ^M yourself if you want the command to be executed.

If there is a second comma, the letter after that comma will be the keyboard shortcut for that menu item. (This will be case sensitive some day, use lowercase).

1.14 rback

Whenever you want to start a program in the background, you can, instead of

```
rback foo
```

just type

```
foo&
```

You can also chose the command to be used for the background starting using the system variable _rback.
@endnode

1.15 prompt

By setting the _prompt and the _titlebar variable to special strings, you can display all kinds of changing data in the prompt and the titlebar. They will be updated every time you issue a command.

1.16 Differences to AmigaDOS

If you're used to the AmigaDOS shell, remember the following:

- Csh internal commands must be lowercase and can be abbreviated
- You can still use your old scripts, with some limitations. But you better convert the scripts to csh scripts. For simple scripts, csh is downward compatible to AmigaDOS. See
  Scripts
  - You can always get more information on a command if csh.doc is in the current directory or in csh: (you can modify this) and you enter 'man <command>'
- The wild card #? doesn't work. Use *

1.17 Differences to UNIX

If you're used to the UNIX csh, remember the following:

- Amiga csh is not script compatible
- Some commands, e.g. foreach, head, tail, work slightly different
- Only simple history modifiers work
- Variable modifiers don't work at all

1.18 restrictions

The following applies only to the V36 version of Kickstart 2.0: The INTERNAL commands cannot be started. The same is true for the commands in C: if they were made resident using the AmigaDOS 'Resident' command (with cshell's 'resident' they work). Thus, you should disable the INTERNAL residents using the -i0 startup option if you have a V36 Kickstart.

The VDK: handler and Frank Seidel's BootRam-Handler have a bug with setting file dates, so when using the copy command you should try the -d and -p switches, otherwise your file date will be bad. (This is not a shell bug)

If using it with conman you may consider starting shell with the -a switch to turn off shell's command line editing and use conmans instead. You'll lose, however, many shell features like file name completion.

CB-handler (a tool that installs a scrollbar in the CLI window) is not 100% compatible with cshell. The log will not always represent the real screen contents.

1.19 startup

Csh can be started in the following ways:

```
csh [-abcCfiknstv] [-c command;command]
csh [-abcCfiknstv] [batchfile1 ... batchfileN]
```

```
 -a  AUX: mode. No command line editing and text highlighting
 -b  starts shell in background, which means only task priority -1.
 -c  allows execution of one command line and then exits out
     of  shell.  This  is  useful for running an internal shell
     commands in the background or from an external application:
  run csh -c "dir df0:; copy -r df0: df1: >nil:; echo Done"
 -C  same as -c, but the command line is not parsed twice. This
     allows passing of file names with blanks within.
  run csh -C rm "Ram Disk:tempfile"
 -f  starts shell in foreground, which means only task priority 1.
     you might reset this prioritiy to 0 at the end of your .login
 -i0 disables INTERNAL residents. For V36 kickstarts.
 -k  sets _nobreak before doing anything
 -n  suppresses starting of s:.login
```

```
 -r  copies the Amiga resident list to the ARP resident list. You
     can't remove them anymore. No copying when under kick 2.0 or
     if ARP residents present.
 -s  globally enables the asterisk * as alias for #? in AmigaDOS 2.0.
     This means you can use * inside file requesters as well.
 -t  terminal mode. You can use command line editing and text high-
     lighting on a VT100 compatible terminal. To swap backspace
     and DEL, refer to the 'keymap' command
 -v  sets _verbose to 'hs' before doing anything.
```

1.20 editing

```
                The  command line can be up to 255 chars.  For a permanent
reminder   of   most  editing  commands,  run  the  script
'menu.sh'  which installs an inutuition menu that contains
edit functions.

MOVING
Left Arrow    One character left
Right Arrow   One character right
Shift-Left Arrow  One word left
Shift-Right Arrow One word right
ESC-Left Arrow    Beginning of line (^A) (^Z)
ESC-Right Arrow   End of line       (^E)

DELETING
Backspace   Previous character
Del     Character under cursor
ESC-Backspace   Previous word     (^W)
ESC-Del     Next word
ESC-x-Backspace   To start of line  (^B)
ESC-x-Del   To end of line    (^K)
ESC-d     Entire line       (^X)

HISTORY
Up Arrow    Recall previous commands, see
               History
                Down Arrow    Recall commands
Shift-Up Arow   Get history from partial (or number)
Shift-Down Arrow  Go below last command of history
ESC-Up Arrow    Get start of history
ESC-Down Arrow    Get end of history
ESC-!     Get history from partial (or number)
^T      Insert tail (all but first word) of previous line
^P      Duplicate previous word (useful for mv)

COMPLETION
TAB     Inserts first matching file name, see
              Completion
                Shift-TAB   Inserts longest common substring
ESC-TAB     Inserts all matching file names (also ESC-*)
ESC-c     Does a quick cd on left word (TAB for cycling)
ESC-~     Inserts the last current directory
^D      Shows all files that match a pattern (also ESC-=)
```

```
EXECUTING LINE
Return      Executes line
ESC-Return    Executes this line of history & brings up next one
^N      Next line. Don't exec this one but store history
^\      EOF (directly exits)

MISCELLANEOUS
^L      Retype current line.
^O      Echo a ^O
^R      Repeat last command (don't play with this)
^U      Undo/Redo last edit
ESC-i     Toggle Insert/Overwrite
f1-f10      Execute preset function key. See
               FKeys
                F1-F10      More commands (Shifted f keys).
Help      Invokes help command

The CTRL keys FGVY are unset, feel free to map them to any
function (see
               Keymaps
              ).  You can also remap
all preset keys. All edit functions work on a
               Terminal
              .
```

1.21 completion

Whenever the cursor is placed on or directly after an incomplete file name and you press TAB, CShell inserts the first filename (sorted alphabetically) that matches the name part already typed. Any wildcards are allowed here, if none are given, '*' is appended. Immediately pressing TAB again brings up the next file name that matched the substring. Shift-TAB will only insert the as much as is common to all files that matched your abbreviation. If pressed again, behaves just like TAB. ESC-Tab inserts the name of the directory where you would have ended up with a quick cd to that substring.

1.22 fkeys

Function keys insert text to the current position on the command line. They may be terminated with a ^M (return). f1 would be non shifted whereas F1 is shifted.

```
  set f1 dir df0:^M
```

will add the text 'dir df0:<return>' to the current line.

```
  set f1 "dir "
```

would only add 'dir ' you could then enter 'df0:<return>'

1.23 terminal

If you want to use csh's command line editing on a VT compatible terminal, you must do the following:

1. Create a file s:Aux-startup that contains
   csh -t
2. Execute the following command
   Newcli AUX: from s:aux-startup

Then csh should show up on the terminal. See also the _hilite variable for highlighting methods.

1.24 parser

The following is done on the command line before it is passed on to an internal or external command:

; delimits commands. echo charlie ; echo ben.

' ' (a space). Spaces delimit arguments.

"string" a quoted string. Trailing quotes are optional. No further parsing happens inside. Example: echo " Indented!"

$name where name is a variable name: Inserts the value of the named variable. See
Variables
^c where c is a character is converted to that control ↩ character.
Thus, say '^l' for control-l.

\c override the meaning of special characters. '\^a' is a circumflex and an a rather than control-a. To get a backslash, you must say '\'.
Also used to overide alias searching for commands.

\nnn insert character code nnn octal. Do not use values between \200 and \232, as they have special meanings.

* ? Wild cards. * stands for any string, ? for any letter. For details refer to @{ " Wildcards " link wildcards}

>file specify output redirection. All output from the command is placed in the specified file. Note: No blank before the file name allowed.

>>file specify append redirection

<file specify input redirection. The command takes input from the file rather than the keyboard (note: not all commands require input; it makes no sense to say 'echo <charlie' since the 'echo' command only outputs its arguments).

```
| Pipe specifier.  The output from the command on the left becomes
  the input to the command on the right. See
              Pipes
               !x  Insert a previous command, e.g. !1 for the first, !! for the
  last one. See
              History
               # enter comment.  The rest of the line is discarded (note: \#
  will, of course, overide the comment character's special
  meaning)

{e hi;e ho} executes two commands as one, so they can be redirected
  together (see ALIAS command). The trailing curly brace is
  optional. See
              Blocks
               $(foo)  insert the stdout of the command 'foo' at this position ↩
                   of
  the command line. Every line of the output will count as one
  argument. The closing parenthesis is optional.

`foo` insert the stdout of the command 'foo' at this position of
  the command line. Every blank separated word will count as
  one argument. Leading, trailing and multiple blanks will be
  removed. The trailing backtick is optional.

--  stop option parsing here. Works for internal commands only.
  Example: rm -- -x will remove the file '-x'


After all this, the parsed command line is passed to
              Execution
             .
```

1.25 wildcards

```
               Most  shell  commands  will accept multiple arguments that
can  be  as  a  result  of wild card expansion.  Also when
calling  an  external  command,  csh will first expand any
wild cards to separate arguments.  If you wish to have the
external  command  handle  it's  own wild carding you will
need  to  insert  quotes  around  the  special  wild  card
characters  or use a special alias (see
              Aliases
             ).
Example:

  arc a new.arc *.txt - shell will expand and pass to arc
  arc a new.arc "*.txt" - let arc expand the wild cards.
  alias arc "*a arc $a" - now shell will never expand

Wildcards allowed:

? match any single character
* match any string
.../* recursive search down ALL sub directories from current level
```

```
~ exclude pattern matching specifier
! synonym for ~, supported for compatibility
& prefixed to patterns, ask confirmation for each file
[]  character class
~ the previous current directory (if separated)

Note that a pattern must contain a '?' or a '*', otherwise
the   other   special   characters   are   not   recognized.
Furthermore,  you cannot specify a single '?' as a pattern
in the first argument to a command, as this will be passed
on  to  the  command  in  order  to  show  its  usage.  If
pattern.library  is  present it LIBS:, it will be used for
the matching.

Examples:

df0:.../* all files in all directories on df0:
df0:.../!*.info full directory tree of df0: but exclude
    any ugly .info files.
!*.o !*.c will result in ALL files matching since what
    doesn't match the !*.o will match the !*.c
df1:&*    all files in root of df1:, but ask
    confirmation for each
*.[co]    all files ending in .c or .o
~*.[co]   all files NOT ending in .c nor in .o
~   the previous current directory
~/*.c   all .c files in the previous current directory
~//   the parent of the previous current directory
.   the current directory
./foo.c   the same as foo.c
..    the parent of the current directory
../foo.c  the file foo.c in the parent directory

Note that some commands prevent wild card expansion. These are:
  - dir, rpn, whereis, window
Those commands will expand the wild cards themselves. This is why
  dir @without( *.? , *.o )
will not work. Instead use:
  set arg @without( *.? , *.o );dir $arg

The  following  symbols are not yet supported by wild card
expansions, but are accepted in search -w and @match( ):

( | ) OR matching
# 0 or more times the pattern following

Examples:

"k#a"   matches ka, kaa, kaaa, etc.
"hel(lo|p)" matches hello or help.
```

1.26 Pipes

```
A  pipe means that the output of one command becomes the input
if another.
```

```
  alias | qsort | truncate
```

will show all aliases, sort them, cut long lines and print the result to the screen.

Pipes have been implemented using temporary T: files. Thus, you should be careful when specifying a 't:*' expansion as it might include the temporary files. These files are deleted on completion of the pipe segment. A nice example of a pipe:

```
  echo "echo mem | csh" | csh
```

Now figure...

1.27 history

History means that whatever commands you type in at the prompt are stored for later retrieve. To see what you typed before, use the 'history' command. There are various ways how you can get your text back.

One way is doing a history retrieve command. The following commands are recognized:

```
!! Execute previous command
!3 Execute third command
!-2 Execute second-but-last command
!hi Execute last command that starts with 'hi'
```

The other way to retrieve old commands is the command line editing. Cursor up and cursor down keys will move through history. You can also get a specific history line into the command line: Type any of the above strings without the first exclamation mark and type shift-cursor up. For example 1 and a shift-up will bring the first command you typed pack into command line editing. Especially useful is typing an abbreviation of the command line you want back and press shift-up until you've got the line you want.

1.28 Command execution

The first argument of the command line is interpreted as the command. Here is how csh tries to execute it:

1) The alias list is searched for an alias with an exactly matching name. Case is significant.
2) Internal commands list is scanned for a command even partially matching name (so you can, for instance, say resi for resident; however, you should specify enough of a command to be unique). Again, case is significant.

3) Then, the list of functions is scanned for a command that matches completely. If one is found, the result of the function is echoed to stdout.
4) Now the command is assumed to be external. Arguments with blanks, semicolons or empty strings will be surrounded by quotes.
5) If the file is a directory, a 'cd <file>' will be performed to it.
6) AmigaDOS and ARP resident list are scanned for it (you can use Shell's 'resident' command to add/remove a file in the ARP list).
7) If the file is in the current directory and it's executable, it is started.
8) Then it is searched in the AmigaDOS path and c: (NOTE: Path assigns to C: under Kickstart 2.0 don't work; use 'path')
9) Now, the shell path ($_path) is searched. If it's found and executable, it's started. If it has the 's' bit set, it will be executed using the appropriate shell. See FOREIGN SHELLS
10) If there exists a file with the suffix '.sh' and the same root in the current directory or in the shell path, it is 'source'd.
11) Then the variable _rxpath is examined. If there exists a file with the suffix '.rexx' and the same root in the current directory or in '$_rxpath', 'RX <file>' will be performed.
12) If all failed, an 'exec' action is sent to the file. See chapter XIV for more info on classes and actions.

To enforce that the external 'dir'-command is used, enter 'Dir'. It is a good habit to uppercase the first letter of all external commands, even if this is not necessary.

1.29 commands

Csh has more than 100 built in commands. Making them built in makes script faster, allows you to operate without a system disk, and removes the need to make lots of small commands resident first. The drawback is the size of csh, but I think it's still acceptable.

Even if you know what on of below commands is doing, check the csh version anyway. It might have more features. If a command has a more UNIX like synonym, it is shown in parentheses.

alias	creates short aliases for command sequences
assign	assigns a logical name to a physical one
type (cat)	write a text file to the screen
cd	changes the current directory
copy (cp)	copies one or more files
dir (ls)	shows the directory
echo	prints a line of text
foreach	repeats command with different args
help	shows all csh commands
info	shows information about a device
input	inputs a variable from stdin
man	shows the manual page for a command
mkdir (md)	creates a new directory
rename (mv)	changes the name of a file

```
 delete (rm)   removes a file or directory
 run           starts a command in the background
 search        searches a text file for a certain string
 set           assigns a value to a variable
 source        starts a csh script

Click to see a  Table of commands
```

1.30 functions

Functions are a special kind of built in commands. They are not (yet) user definable. Since they're used mostly in scripts, they can not be abbreviated. They all do small tasks and yield output for csh to use. There several ways to use functions as shown in the 'abs' function:

```
  echo @abs( -5 )    --->  5
  abs -5             --->  5
  $(abs -5)          --->  5
```

If using the first form, the function call will be replaced by their return value(s). The functions must be preceded by a blank and a blank must follow the opening and precede the closing parenthesis. There must be no blank between the function name an the opening parenthesis.

Later versions of Shell might allow that functions need be at the beginning of an argument, so quote any @-signs not used for functions.

Functions may be nested. The function names themselves are case sensitive, but the operations (like strcmp) aren't.

When using functions as commands (the second form), remember that the list of functions is scanned only after the list of commands.

The third form is nothing function specific, this works with any command, see

Parser

Click to see a Table of functions

1.31 variables

Wherever $xxx occurs in a command line, csh checks for a local variable xxx and inserts its value if it exists, otherwise csh looks for a normal variable xxx. If that fails as well, csh looks for an ENV: variable xxx and

```
inserts  its  value  if  it exists.  Finally, if even that
fails, "$xxx" is inserted (this behaviour may change).

Variable   names   can  consist  of  0-9,  a-z,  A-Z,  and
underscore  (_).   There  is no limit on the length of the
string stored in a variable.

Examples:
  set name Fred
  echo Hello $name     ---> Hello Fred

  setenv a ENV:-value
  set    a global-value
  local  a
  set    a local-value
  echo $a              ---> local-value
  unset a; echo $a     ---> global-value
  unset a; echo $a     ---> ENV:-value

Variables can store multiple words, some kind of arrays.
  set a  1 2 3 ; words $a     ---> 3
  set b "1 2 3"; words $b     ---> 1
  set c "1 2" 3; words $c     ---> 2
You  won't  see  a difference between $a, $b and $c if you
just  echo  them,  but  in  a foreach, the cause different
number of runs.

There  are  a  number  system  variables.   They are write
variable  that  have  a  side  effect on your system (e.g.
changing  the  title  bar),  and  some  others,  the  read
variables,  that tell you something about your environment
(e.g.   the  current shell version).  You can also overlay
the  write  variables with a local variable, so any change
only  takes  place  while  the  current  context is valid.

Click to see the  System Variables
```

1.32 programming

```
                One of the main reasons why I mostly prefer a command line
interface  over a GUI is that you can very easily automate
repeated  command  sequences.  And one of the main reasons
why  I  always  preferred csh over the other shells is its
scripting language.

There  are  two  mechanisms  how  you  can program command
sequences:  Aliases  and scripts (batch files).  They can
do  very  similar  things,  the  main difference is that a
script  is  always  one  file,  while  many aliases can be
defined  within  one  script. Aliases are usually used for
short command sequences, scripts for long ones.


               Aliases
                memory-resident command sequences
```

```
Scripts
 disk file command sequences

Blocks
 compound statements

Exceptions
 error handling
```

1.33 aliases

```
 Aliases are one of the most powerful and most used features of ↩
    csh.
Let's start with a simple one:

  alias dc DiskCopy

From  now  on, you can type 'dc' wherever you'd have typed
DiskCopy  before,  e.g.  yoy may issue 'dc df0:  to dh1:'.
This  alias  will  be forgotten next time you boot, so you
best put this line in your s:login.sh.


Next level:  Arguments included.  Sometimes you'll want to
insert  the  string that was passed to the aliased command
at a different location than appended to the end.

  alias rm "%f echo -n \"Sure? \";input a;if $a = y;rm $f;endif

Here, when you issue a 'rm', the argument(s) are stored in
the local variable f.  Then the command line is evaluatet.
Note  the  backslashes  in  front  of  the  quotes inside,
they're  needed to prevent the quotes from terminating the
very  first quote.  Don't forget that this command line is
parsed  twice:   Once  when  doing  the  'alias'  command,
removing  the  outer  quotes,  and once when executing the
alias.   To  check  how  the alias you just defined really
looks, type 'alias rm':

  rm         %f echo -n "Sure? ";input a;if $a = y;rm $f;endif

Note we have called 'rm' from the alias. Aliases can call
each other, but direct recursions are prevented (except
inside blocks, see below)



The next level is several arguments.  We want to teach the
'pri'  command  to accept process names as well as process
numbers:

  alias pri "%task%prio pri @clinum( $task ) $prio

In  case  this alias is called with more than 2 arguments,
```

the last variable gets all the rest.

Very common is the use of 'foreach' in aliases. With it, you can give programs that can only handle one argument at a time the capability to handle any number of arguments.

```
alias tg "%a foreach i ( $a ) \"Turbogif $i
```

A special feature of the alias execution is that if you use * instead of %, wild card expansion will be suppressed for this alias, so you can pass * ? and the like to the programs like normal characters. LHA, for example, does the wild card expansion by itself.

```
alias lha "*a lha $a
```

What happens if you have defined

```
alias sc  "search *.c
```

and you call this alias with 'sc -c MyFunc'? The -c option will be interpreted as a file name. The solution:

```
alias sc  "%a search @opt( $a ) *c @arg( $a )
```

Finally, it is possible to give a block of commands instead of a quoted command line. This could look like this:

```
alias rm {%f echo -n "Sure? ";input a;if $a = y;\rm $f;endif
```

Note we don't need to escape the quotes anymore, but we have to escape the 'rm' command to prevent recursive calling of the alias 'rm'. Blocks can be spread over several lines in script files. See
Blocks
for more.

Aliases may be arbitrary long and receive an arbitrary number of arguments.

Some aliases are predefined whenever you start a new csh. These are:

cdir
Use "cdir directory" to clear the screen, set CD to directory, and list it.

cls
Simply clear the screen.

dswap
Exchanges current and the previous current directory. For use in scripts as the symbol for last current directory may change.

exit
Leave Shell and exit CLI.

manlist
Display a list of possible arguments to man.

rx
Executes a REXX script. Prevents unwanted starting of 'rxrec'.

1.34 scripts

Scripts are text files, where each line is a valid csh command. Scripts are used for two things: Doing large programs in csh, and doing things you'd like to be done on startup. There are some commands that only work in scripts (at the moment), for example 'label' and 'goto'. Scripts are executed using the 'source' command.

Scripts files would usually receive a .sh suffix and NOT have their s-bit set. Now if such a file is anywhere in your $_path, it gets invoked. Example:

```
mkdir ram:foo
echo >ram:foo/bar.sh "echo Hello World!"
set _path ram:foo
bar
       ---> Hello World!
```

A very good place to put your scripts is the CSH: directory which is in $_path by default.

Other shells' scripts are supported as well. If a file has the 's' bit set, csh reads the first line. If this line start with /* , it is assumed to be a REXX script. If it starts with #! or ;! , the rest of the line will be the command used to execute the script. If none of the above is true, it is assumed to be an AmigaDOS script and c:Execute will be used to execute it. Example beginnings for Sksh and csh (you need the latter if you don't like the .sh extension):

```
#!SKsh -c source
#!csh -c source
```

The maximum line length on a script is 512 bytes. If you want more, you can concat lines by adding \ to the very end of your line, so the next line will be appended to

this one.

If the last character of a line is {, all following lines will be appended to this one until the matching } is found at the very end of a line. Example:

```
  foreach i ( a b c ) {
    foreach j ( 1 2 3 ) {
      echo "*****
      echo $i
      echo $j
    }
  }
```

The arguments passed to a script are stored in $_passed. Note this variable is global, you might want to store a local copy somewhere.

For examples on how to write complex scripts, see demo.sh

1.35 blocks

Blocks have several functions: They create a context for local variables, they can be used to redirect groups of commands, they can spread over several lines in scripts, they can replace aliases that are only used in one place, and they simplify nesting a lot.

Local context:

```
  alias foo "local x; set x hi
  foo
  alias bar {local $y;set y hi
  bar
```

After the foo command, your current context (e.g. the script this alias is issued in) will have a local variable x which could be altered by mistake by other aliases, but after the bar command, nothing has changed in the world outside the alias.

Redirection:

```
  {echo "Contents of bar.txt:";cat bar.txt} | more
```

This will redirect both the echo and the cat command to more.

Multi line blocks:

```
  foreach i ( a b c ) {
    foreach j ( 1 2 3 ) {
      echo "*****
      echo $i
      echo $j
    }
  }
```

These lines in a script will be concatenated, leaving in line end marks. They'll all count as one single argument to the first 'foreach'. There's no limit on the size of

such a block.

Alias replacement:
  class gif action view={%a foreach i ($a) "Turbogif $i
Here, an alias with an argument gets defined, but it's forgotten every time after execution.

Nesting:
  fornum a 1 2 "fornum b 1 2 \"fornum c 1 2 \\"echo \\\\"#
  fornum a 1 2 {fornum b 1 2 {fornum c 1 2 {echo "#
These tw lines to the same. As soon as csh sees the opening brace, it stops parsing until it finds the matching closing brace. This means you don't have to escape (but a \ in front) anything inside, just type the commands as if you wanted to execute them from the command line.

1.36 exceptions

If no _except variable exists, any command which fails causes the rest of the line to abort as if an ABORTLINE had been executed. If the _except variable exists, it is of the form:

"nnn;commands..."

where nnn is some value representing the minimum return code required to cause an error. Whenever a command returns a code which is larger or equal to nnn, the commands in _except are executed before anything. WHEN _except EXISTS, THE COMMAND LINE DOES NOT ABORT AUTOMATICALLY. Thus, if you want the current line being executed to be aborted, the last command in _except should be an "abortline".

Exception handling is disabled while in the exception handling routine (thus you can't get into any infinite loops this way).

Thus if _except = ";", return codes are completely ignored.

Example:

set _except "20;abortline"

1.37 classes

File classes are good for two things: Identifying files and command overloading. The latter means that the same command with files of different type performs completely different actions.

Identification
or how to recognize a file type

1.38 identification

You can define a class of files using several 'class' commands.
Here a simple example:

```
class picture  suff=.pic suff=.iff suff=.ilbm
class anim     suff=.anim
```

From now on, everything with the suffix .pic, .iff or .ilbm will
be identified as a picture. Please note that there may be no blanks
between the names and the '=', and that blanks inside the names
must be put in quotes. So these are the ways to identify a file:

```
suff=.doc        True if the suffix of the file is .doc
name=readme      True if the file is "readme"
name="mod.*"     True if the name starts with 'mod.'
offs=14,DC..C4FD True if the bytes starting at $14 are $DC,
                 anything, $C4, $FD  (all numbers hexadecimal!).
                 Each pair of dots means one byte ignored.
chars            True if 90% of the bytes in the file are 32..127
                 or 9..13
default          Always true, used to define the default type
```

Note that only the first character is examined, so 's' = 'suff'.
One class can be identified by more than one 'class' statement.
They are looked at in the same sequence they were entered. So to
make sure that an zoo archive misnamed as .lzh is identified
correctly, use the following 'class' statements:

```
class zoo offs=14,DCA7C4FD
class lzh offs=2,2D6C68..2D
class zoo suff=.zoo
class lzh suff=.lzh
```

Moreover, there is a builtin class 'dir', which means directory.
Now we know many file types. But what to do with them? This is
where we define 'actions'.

1.39 actions

There may be one or more 'class' commands that define what actions
need to be taken in various cases for that specific class:

```
class zoo actions view="zoo -list" extr="zoo -extract"
class lzh actions view="lz l"      extr="lz e"
```

Whenever somebody tries to 'view' a test.zoo, the command 'zoo -list test.zoo' will be issued, but if he tries to view test.lzh, then 'lz l test.lzh' will be executed. Note that any command supplied here goes through the normal csh parser, so AmigaDOS and csh paths will be searched. Aliases with arguments are allowed here, too, so whatever the user typed will be stored in the variable after the '%'.

How do I tell a file that I want to 'view' it? There comes the second command used for object oriented features:

```
action view test.zoo
```

will first identify the type of that file and then apply, if possible, the 'view' action to it. Of course, this works best inside an alias: alias v "action view" will define a v-command that views all types of files known to cshell. Similarly, you can define alias xtr "action extr" and use this command to extract files from any type of archive.
There is one action that will be sent to every file that you try to start but is not executable. This action is 'exec'. Assume you have defined the class 'picture', then after

```
class picture actions view=ShowIFF exec=ShowIFF
```

you can display a picture using Mostra by just typing its name. More builtin actions like 'rm' and 'dir' may be implemented, so don't use command names for action names.

The batch file class.sh defines a few useful classes.

1.40 superclasses

Assume you have a class for .c files, one for .h files, and one for .asm files. You might want to make the difference between them when identifying them, but in the end, they're all ASCII, aren't they? You can stat this with the command

```
class c_source suff=.c is=ascii
```

Now whenever an action on a file of the type c_source fails, the file is interpreted as of type ascii, and the same action is attemted again. This goes on until a class has no more superclass.

1.41 keymaps

You define a keymap as a collection of key/function pairs using the 'keymap' command. Both are given as numbers. There can be several keymaps which activate each other, but at first we only edit keymap 0, which is active at the

```
beginning.   All keys you define will eventually overwrite
the  old  definitions  in  an existing keymap.

Examples:

  keymap 0    66=49    # the  B key will beep
  keymap 0     2=16    # ^B key will erase line
  keymap 0  1122=35    # ESC-b will show matching files
  keymap 0 9=31 521=30 # Swaps TAB and SHIFT-TAB

@{ " Key codes     " link keycodes    } : How to specify a key
@{ " Edit funcs    " link editfuncs   } : How to specfiy an editing operation
@{ " Command types " link commandtypes} : How to specify other funcs
```

1.42 keycodes

```
1..255    The corresponding ASCII character
256   Up Arrow
257   Down Arrow
258   Right Arrow
259   Left Arrow
260   Help
261..270  F1..F10  (unshifted)


Modifiers (add them to the key code)

512       SHIFT (mainly necessary for arrows and fkeys)
1024      ESC   (was pressed & released before this key)
```

1.43 editfuncs

```
- Movement      Move cursor...
 0  CursLeft    1 left
 1  CursRight   1 right
 2  WordLeft    1 word left
 3  WordRight   1 word right
 4  BegOfLine   to beginning of line
 5  EndOfLine   to end of line

- Deleting      Delete...
10  Backspace   char left from cursor
11  Delete      char right from cursor
12  BkspcWord   word left from cursor
13  DelWord     word right from cursor
14  DeleteToSOL to start of line
15  DeleteToEOL to end of line
16  DeleteLine  whole line

- History insert
20  Back        Move one line back in history
21  Forward     Move one line forward in history
```

```
22  Beg         Move to first line in history
23  End         Move to last line in history
24  Complete    History retrieve like '!'
25  Exec        Execute history line & bring up next
26  Tail        Insert previous line except first word
27  Bottom      Go below last history command
28  DupWord     Duplicates the last word on this line

- Completion
30  Normal      Insert first matching file (or cycle)
31  Partial     Insert common substring of all matching files
32  All         Insert all matching files
33  Directory   Find dir in quick cd list
34  LastCD      Insert last current directory
35  Show        Shows all matching files

- Special
40  Insert      Toggle Insert/Overwrite
41  Quit        Silently perform 'quit'
42  Help        Silently perform 'help'
43  Refresh     Redraw current line
44  Execute     Execute current line
45  Leave       Edit new line, store this in hist
46  EOF         Terminate shell
47  NOP         Do nothing
48  Echo^O      Echoes a ^O
49  Beep        Echoes a ^G

- Other
50  Fkey        Execute command associated to last fkey
51  Menu        Execute command associated to last menu
52  Undo        Undoes last edit
53  Repeat      Repeats last function
54  SwapChar    Swaps the two chars left of cursor
```

1.44 commandtypes

```
Command types

0   +x    Editing function x, see above descriptions
512 +x    Setmap x,     x=0..7
1024+x    Insert key x, x=1..255
1536+x    Macro x       x=1..15         (unimplemented)
2048+x    String x      x=1..15         (unimplemented)
```