

Intermediate Unix Skills on MEnet Systems

Version 1.2

R W Kaszeta

August 16, 1999

Contents

1	Shell Scripts	1
1.1	csh vs. sh	1
1.2	Creating scripts	2
1.3	csh scripts	2
1.4	sh scripts	4
1.5	Debugging Shell Scripts	5
2	Account Customization	6
2.1	Shell Customization	6
2.2	X Startup Scripts	8
2.3	X Defaults	9
2.4	Other Account Configuration	9
3	Useful Programs	10
3.1	Make	10
3.2	Diff/Patch	10
3.3	ssh	11
3.4	Other Useful Programs	11
4	Unix Tips and Tricks	12
4.1	xterm Tricks	12
4.2	Emacs	12
5	Pointers to further information	13

Introduction

This course is meant to provide users that are already familiar with MEnet Unix workstations with knowledge and examples of some of the more advanced features of the Unix operating systems.

People taking this course should either be experienced users in a Unix environment, or at least be familiar with my “Basic Unix Skills” course, <http://www.me.umn.edu/courses/>

©1999 Richard W Kaszeta

¹For users from a DOS/Windows background, a shell script is basically the same thing as a “batch file”.

[shortcourses/UNIX/basicunix.pdf](#).

We encourage users to give us feedback and recommendations for improving this document. Please mail suggestions to me (kaszeta@me.umn.edu) or MEnet (menet@me.umn.edu).

1 Shell Scripts

Whenever you login to a Unix system you are placed in a program called the shell. It is called the “shell” because it sits between you and the actual operating system (usually called the “kernel”), providing a user interface. The shell serves a number of functions under Unix, including

Interactive Use: The shell provides a command-line interface to the operating system for the user.

Customization: The shell provides a method for users to customize your Unix environment (for example, where to look for programs, and some simple program configuration). We’ll talk about this more in section 2.1.

Programming and Task Automation: The shell also provides a series of commands that can be used to create programs called “shell scripts.”¹ The shell even provides for repeated execution of commands (like “while” loops) and condition execution (“if-then-else” loops), so you can write programs that allow you to automate a lot things you might want to do in Unix.

1.1 csh vs. sh

As mentioned previously, the shell provides a command-line user interface to the operating system. On most Unix systems there are a number of shells available, which provide slightly different user interfaces. The original Unix shell, the Bourne Shell, is provided on every Unix machine, and is called `/bin/sh`. An enhanced version of this shell with many additional features, the Bourne-Again Shell, or “bash”, is also available as `/bin/bash`.

Additionally, an alternative to the Bourne Shell was created to facilitate shell programming using a C language-like syn-

tax. The result is the “C Shell”, available as `/bin/csh`. Like `bash` is provided as an enhanced version of `sh`, on MEnet Unix systems, an enhanced version of `csh` called the “Tenex C Shell”, or “`tcsh`” for short. It is available on MEnet systems as `/bin/tcsh`.

To become fluent in shell programming, you must learn the appropriate syntax for your shell. Both `sh` and `bash` use the same language syntax (which I’ll call “sh syntax” from now on). Similarly, both `csh` and `tcsh` use the same syntax, which I’ll call “csh syntax”.

1.2 Creating scripts

The first step in creating a script is to place the commands you want to execute into a file that you’ve created with an editor. The first line of script always looks like the following:

```
#!/bin/csh
```

The `#!` characters tell the operating system to treat the program as script instead of a simple text file, and the characters following the `#!` tell the operating system what interpreter is supposed to process the commands in the file (in this case, `/bin/csh`, so this is a `csh` script). Similarly, if you were creating a `bash` script, it would instead start with `#!/bin/bash`.²

The second step is to make sure that the resulting file is labelled as an executable file using `chmod`. So if my script file is called “myscript”, I can make it executable by saying

```
chmod +x myscript
```

at the shell prompt (see the `chmod` man page for more details).

Now you know how to make a script file and let the operating system run it, so all we have to do is tell you what to put in these scripts.

1.3 csh scripts

The simplest scripts to write are C-shell scripts, since they use the exact same syntax as the default user command line (which for most users is `/bin/tcsh`). So you can create simple C-shell scripts just by starting a file with `#!/bin/csh` and then listing out the commands you would type if doing each step by hand. Lines that begin with `#` are comments and not executed by the shell. For example, if you want to run your program (called `a.out`), print the output to the printer, and then clean up and delete the output file, the script would look like:

```
#!/bin/csh
a.out > file.out
lpr -Pme155 file.out
rm file.out
```

1.3.1 Variables and Quotes

`csh` allows the user to create variables. This done using the `set` command, with the syntax,

```
set var=value
```

Similarly, a variable can be an array of strings using C-style syntax. For example, I can set the variable “mylist” to a list containing “one”, “two”, and “three” using

```
set mylist=("one" "two" "three")
```

Once a variable is defined, the value of the variable is accessed by placing a `$` in front of it³. So as an example, the following code sets the variable “fred” to the value “barney”, and then print it out,

```
set fred="barney"
echo $fred
```

Basically, the `$var` operator acts as a “substitution” operator, replacing whatever is after the `$` with the contents of the variable `var`. A number of additional substitutions are possible:

<code>\${var}</code>	The value of variable ‘var’
<code>\${var[i]}</code>	The <i>i</i> th member of the variable ‘var’. <i>i</i> can also be a range, such as <code>m-n</code> (the <i>m</i> th through <i>n</i> th members), <code>-n</code> (the 1st through <i>n</i> th members), <code>m-</code> (the <i>m</i> th through last members), or even <code>*</code> , all members.
<code>\${#var}</code>	The number of words in the variable ‘var’
<code>\${#argv}</code>	The number of command line arguments supplied to the script
<code>\$0</code>	The name of the script
<code>\${argv[i]}</code> , or <code>\${i}</code>	The <i>i</i> th command line argument
<code>\$*</code>	Same as <code>\$argv[*]</code>
<code>\${?var}</code>	1 if ‘var’ is set, 0 otherwise
<code>\$\$</code>	The PID of the running script
<code>\$!</code>	The PID of the last background process
<code><</code>	Read a line from standard input

Note that the `$` operator works inside of double-quoted (but not single-quoted) text strings and back-ticks, so something like

```
set fred=barney
echo "Fred is set to $fred"
```

will work.

There is one more form of substitution, and that is “back-tick” substitution. If you place a text string in back-ticks, (i.e. ‘`cmd`’), the shell will run `cmd` and replace the back-tick expression with the results of this command. This is useful for

²You’ll see later that in some cases this works with programs that aren’t shells. For example, `perl` and `make` scripts can be written in this manner.

³Note that environmental variables, such as your Unix `PATH`, can be accessed this way as well

both scripting and interactive use. A good example would be `grep` for files containing the word “reward” sorted by date:

```
grep -i reward `ls -t`
```

1.3.2 Expressions

Expressions are used in many shell constructs, such as conditional (`if`) statements and looping (`foreach` and `while`) statements. Expressions can perform comparison, assignment, and file testing operations. The syntax for creating expressions is basically the same as the C language. Some of the basic useful operators include:

Comparison Operators:

<code>== !=</code>	Equals, not equal
<code><= =></code>	Less than or equal to, Greater than or equal to
<code>< ></code>	Less than, greater than
<code>=~</code>	String on left matches a filename pattern on the right containing *, ?, or [...]

Assignment Operators:

<code>=</code>	assign value
<code>+= -=</code>	assign value after addition/subtraction
<code>*= /= %=</code>	assign value after multiplication/division/modulus
<code>&= ^= =</code>	assign value after bitwise AND/XOR/OR
<code>++</code>	increment
<code>--</code>	decrement

Arithmetic Operators:

<code>* / %</code>	Multiplication/Division/Modulus
<code>+ -</code>	addition/subtraction

Logical Operators:

<code>~</code>	Binary Inversion
<code>!</code>	Logical Negation
<code><< >></code>	Bitwise Left Shift, Bitwise Right Shift
<code>&</code>	Bitwise AND
<code>^</code>	Bitwise XOR
<code> </code>	Bitwise OR
<code>&&</code>	Logical AND
<code>!!</code>	Logical OR
<code>+ -</code>	addition/subtraction

File Test Operators:

<code>-d file</code>	Tests true if 'file' is a directory
<code>-e file</code>	Tests true if 'file' exists
<code>-f file</code>	Tests true if 'file' is a plain file (as opposed to a directory or special device)
<code>-o file</code>	Tests true if the user owns 'file'
<code>-r file</code>	Tests true if the user can read 'file'
<code>-w file</code>	Tests true if the user can write to 'file'
<code>-x file</code>	Tests true if the user can execute 'file'
<code>-z file</code>	Tests true if 'file' has zero size
<code>!</code>	Negates the results of any of the above tests

1.3.3 Conditional Execution, the `if` command

The `if` command allows you to test an expression, and if that expression is true, run the command. If the expression is false, ignore it. The syntax for the `if` command is

```
if (expr) then
  command
  command
endif
```

For example, if we wanted to test to see if we are on a Linux machine (by using the `uname` command, which returns the OS type) and then print out a statement if we are, the `if` statement would be

```
if ('uname' == "Linux" ) then
  echo "We're running under Linux"
endif
```

An `if` statement can also include an “else” clause, code which is executed if the test statement fails. The syntax is

```
if (expr) then
  command
else
  command
endif
```

So we could extend our above example to print if we’re *not* running under Linux:

```
if ('uname' == "Linux" ) then
  echo "We're running under Linux"
else
  echo "We're NOT running under Linux"
endif
```

1.3.4 Doing things multiple times: `while` and `foreach`

`csh` provides two looping constructs. The `while` command has the syntax

```
while (expression)
  commands
end
```

which will run the listed commands as long as the expression tests true. An extreme example is to give “1” for the expression, since this (by definition) always tests true. This allows you to loop indefinitely. So to run a command roughly once every minute⁴:

```
while (1)
  rm outputfile.old
  mv outputfile outputfile.old
  command > outputfile
  sleep 60
end
```

⁴The `sleep` command pauses the shell for the specified number of seconds

More useful is the `foreach` command, with the syntax

```
foreach name (wordlist)
  commands
end
```

For every word in “wordlist”, the shell sets the variable “name” to that word and runs the command. One use of this is to run a command on each of a number of files (in this case, use `convert`⁵ to convert all the `.gif` files in a directory to `.jpg` files). Note that this example uses back-tick expansion and the `basename` command, which removes the extension from a filename:

```
foreach file (*.gif)
  convert $file `basename $file .gif`.jpg
end
```

1.3.5 Other `cs`h features

`cs`h has a wide variety of additional programming constructs, which are too extensive to cover here. If you are interested, there are a number of additional `cs`h resources, including

1. The `cs`h or `tc`sh man pages.
2. The `cs`h chapter in either *Unix in a Nutshell* or *Linux in a Nutshell* (which covers a lot of `tc`sh- and `ba`sh-specific techniques in addition to the standard topics), both by O’Reilly and Associates. Available at most bookstores for about \$20.
3. *Using cs*h & *tc*sh, also by O’Reilly and Associates.

1.4 `sh` scripts

The Bourne Shell (and its cousin Bash) is the other predominant flavor of shell present on Unix systems. It provides a similar feature set and a similar syntax to `cs`h, but is different enough to be covered separately

1.4.1 Why another shell language?

At this point, since I’ve already taught you basic C-shell syntax, you’re wondering why we’re learning another shell as well. There are many reasons including:

1. You can *always* depend on `/bin/sh` or a compatible shell being present on a Unix system, whereas `cs`h is optional and may not always be present.
2. Many users prefer `sh` syntax, finding it more intuitive⁶.
3. Many of the `sh` derivatives, such as the Korn Shell (`ksh`) and the Bourne-Again Shell (`ba`sh) provide a number of powerful scripting features not available in C-shell-compatible shells.
4. Occasionally you may need to edit `sh` scripts, so it’s good to know how they work.

1.4.2 Creating `sh` Scripts

`sh` scripts are created in the same manner as `cs`h scripts. First you make sure the first line of the script begins with

```
#!/bin/sh
```

and that the script is marked as executable using `chmod +x`.

1.4.3 Variables and Quoting

Like `cs`h, `sh` has variables. However, you don’t need the `set` command to set variables⁷, the syntax is instead simply

```
var=value
```

Similarly to `cs`h, once a variable is defined, the value of the variable is accessed by placing a `$` in front of it. So as a `sh` version of our previous example, the following code sets the variable “fred” to the value “barney”, and then prints it out,

```
#!/bin/sh
fred="barney"
echo $fred
```

`sh` also has some variable substitution available:

<code>\${var}</code>	The value of variable ‘var’
<code>\${var:-value}</code>	The value of variable ‘var’ if set, otherwise use ‘value’
<code>\${var:=value}</code>	The value of variable ‘var’ if set, otherwise use ‘value’ and set ‘var’ to ‘value’
<code>\${var:?value}</code>	The value of variable ‘var’ if set, otherwise print ‘value’ and exit
<code>\${var:+value}</code>	The value of ‘value’ if ‘var’ is set, otherwise use nothing

`ba`sh makes a number of additional `$` substitutions available, such as most of the valid `cs`h constructs. Consult the `ba`sh documentation for more information.

`sh` also has some built-in variables like `cs`h as well:

<code>\$0</code>	The name of the script
<code>\$n</code>	The <i>n</i> th command line option
<code>\$#</code>	The number of command line arguments
<code>\$*</code>	All of the command line arguments, quoted as a single string
<code>"\$@"</code>	All of the command line arguments, quoted individually as a separate strings, like “\$1” “\$2” “\$3”
<code>\$\$</code>	Current PID of the script
<code>#!</code>	PID of the last background command
<code> \$?</code>	Exit value of last executed command

⁵`convert` is available on the Solaris and Linux machines, and converts image files between a large number of formats

⁶Myself, I prefer `ba`sh when writing shell scripts, for reasons of both syntax and the additional features of `ba`sh. However, I use `tc`sh as an interactive shell.

⁷However, if you are used to it, you can still use the `set var=value` syntax as in `cs`h

1.4.4 Test Expressions

`sh` itself doesn't really have expressions that it can evaluate, but it does have the `test` command which works in a similar manner. The syntax is

```
test condition
```

or

```
[ condition ]
```

where the condition is one of

File Test Conditions:

```
-d file      'file' exists and is a directory.
-f file      'file' exists and is a regular file.
-r file      'file' exists and is readable.
-s file      'file' exists and has a size greater than
             zero.
-w file      'file' exists and is writable.
-x file      'file' exists and is executable.
```

String Conditions:

```
-n s1        String 's1' has non-zero length.
-z s1        String 's1' has zero length.
s1 = s2      Strings 's1' and 's2' are identical.
s1 != s2     Strings 's1' and 's2' are not identical.
string       'string' is not null.
```

Integer Conditions:

```
n1 -eq n2    'n1' equals 'n2'.
n1 -ne n2    'n1' does not equal 'n2'.
n1 -ge n2    'n1' is greater or equal to 'n2'.
n1 -gt n2    'n1' is greater than 'n2'.
n1 -le n2    'n1' is less than or equal to 'n2'.
n1 -lt n2    'n1' is less than 'n2'.
```

All of these expressions can be negated with a `!`. Note that on some systems additional `test` expressions are available, '`man test`' for information. Conditions can also be combined, using either

```
(condition1) -a (condition2)
```

for a logical AND, or using `-o` instead for a logical OR.

1.4.5 Conditional Execution and the `if` Command

`sh`'s `if` command uses a very similar syntax to `csh`:

```
if condition
then commands
fi
```

Thus our example of printing "We're running under Linux" if we are running under Linux would be written as

```
#!/bin/sh
if [ 'uname' = "Linux" ]
then echo "We're running under Linux"
fi
```

Similarly, you can provide an "else" clause:

```
#!/bin/sh
if [ 'uname' = "Linux" ]
then echo "We're running under Linux"
else echo "We're NOT running under Linux"
fi
```

1.4.6 Looping Constructs, the `while` and `for` commands

The `sh` `while` command has nearly identical syntax to the `csh` version:

```
while condition
do
  commands
done
```

Likewise, the `for` command has a similar syntax to the `foreach` command:

```
for name in wordlist
do
  commands
done
```

So to redo our example of using `convert` to convert all the `.gif` files in a directory to `.jpg` files, the code would be:

```
for file in *.gif
do convert $file 'basename $file .gif'.jpg
done
```

1.4.7 The Bourne-Again Shell

On all MEnet systems the Bourne-Again Shell, or `bash` is available. `Bash` is an `sh`-compatible shell that incorporates useful features from other shells such as the Korn shell, `ksh` and the C shell, `csh`. It offers functional improvements over `sh` for both interactive and programming use (particularly additional variable expansion, process control, and test condition capabilities). The best sources of information for `bash` include

1. The online `bash` documentation such as the man pages and info pages.
2. The `bash` chapter in *Linux in a Nutshell* by O'Reilly and Associates, available at your local bookstore.
3. *Learning the bash Shell*, also by O'Reilly and Associates, available at your local bookstore.

1.5 Debugging Shell Scripts

It is important to note here that often when you are writing shell scripts you will make minor mistakes that will require debugging. Both `sh` and `csh` derivatives support invoking the shell with a `-v` option, which causes the script to print out each line of the script file as it is executed. To enable this, simply change the first line of the script from

```
#!/bin/sh
```

to

```
#!/bin/sh -v
```

Additionally, recent versions of the `bash` shell include a `bash` script debugger. See the `bash` documentation for details.

2 Account Customization

Although MEnet tries to provide a useful base configuration for your account, users often wish to customize their MEnet account to add additional features or change the existing setup. In this section we'll talk about some basic methods of configuring your account.

2.1 Shell Customization and Startup Files

As mentioned previously, your shell acts as an interface to the operating system. Any of the programs you run as a user are run by the shell, so a lot of configuration of your account can be done by simply configuring your shell.

In these sections we'll talk about customizing both of the standard MEnet shells, `tcsh` and `bash`.

2.1.1 tcsh Customization

Upon startup, `tcsh` begins by executing commands from some "dot files" in the user's directory. When invoked, `tcsh` first tries to load `~/tcshrc` (or if no `~/tcshrc` file is present, `~/cshrc` is loaded instead). Additionally, if your shell is a *login shell* (meaning you login via `telnet` or `rlogin`, or run `xterm -ls`), it additionally runs commands from the file `~/login`. These shells are C-shell scripts, and can therefore use any of the syntax discussed in 1.3.

A basic `~/tcshrc` or `~/cshrc` file on MEnet systems must be marked executable (i.e. `'chmod +x'`) and start with

```
source /etc/Cshrc
```

This tells the shell to load in the file `/etc/Cshrc`, which contains basic configuration information necessary on MEnet systems, such as setting up the basic `PATH` (what directories the shells search for programs), the `Modules` system, and some useful aliases.

Similarly, a basic `~/login` file must be marked executable (i.e. `'chmod +x'`) and start with

```
source /etc/Login
```

which tells the shell to load in the file `/etc/Login`, which on MEnet systems reports any system news and whether you are exceeding quota.

You can configure your shell by adding additional commands to these files. Usually these customizations always go in

`~/tcshrc`. However, if it is information you only want to see when logging in, an not every time you start a shell or `xterm`, then add the customizations to `~/login`.

Setting Environmental Variables The Unix operating system includes the concept of the *Environment*, a list of variables that can be set in your shell that are passed along to any programs the shell runs. We've already talked about the `PATH` environmental variable, which tells the operating system which directories it should search for programs.

Many programs can be configured by setting "Environmental Variables". This is done by including a line in your startup files that looks like

```
setenv VAR value
```

where `VAR` is the name of the variable (environmental varies are always all-uppercase), and 'value' is the value you wish to assign to `VAR`.

Some useful environmental variables include:

Variable:	Function:
EDITOR	Tells programs what editor they should use by default. Set this to the name of your favorite editor
WWW_HOME	Tells WWW Browsers such as <code>lynx</code> or <code>netscape</code> what URL to load by default (preset to <code>http://www.menet.umn.edu</code> on MEnet systems)
PATH	Tells the operating system where to look for executable programs, using a colon-separated list of directories.
PAGER	Tells programs such as <code>man</code> what program to use to show you a file a page at a time. Many users set this variable to 'less'
MANPAGER	Like <code>PAGER</code> , but only affects <code>man</code> .
PRINTER	Tells <code>lpr</code> what printer to sent print jobs to by default

So, for example, if we wanted to customize `tcsh` to use `emacs` as an editor and add `~/scripts` to the end of your `PATH`⁸, your `~/cshrc` file should look like

```
source /etc/Cshrc

setenv EDITOR emacs
setenv PATH ${PATH}:~/scripts
```

(Note that in this example we are using `${PATH}` to insert the existing value of the `PATH` variable)

Additionally, many Unix programs allow you to specify default command-line switches by using an environmental variable with the same name as the program. For example, the `nenscript` program uses the environmental variable `NENSCRIPT`, so if we always invoke `nenscript` as `'nenscript -2rG -Pme10ads'`, then we could add

⁸Note that on MEnet systems `~/bin` and `~/bin/<osname>` are always included in your default path

```
setenv NENSCRIPT '-2rg -Pme10ads'
```

to our `~/.cshrc` (note that `'-2rg -Pme10ads'` is in single quotes so that `setenv` sees it as a single argument). Many additional programs can be customized like this, consult the man pages of your frequently used programs to see if they support this sort of customization (some other programs that support this include `less`, `mpage`, and `gzip`).

Creating Aliases The `alias` command in `csh` allows users to create shorthand “aliases” for longer commands. The syntax for this is

```
alias name 'command'
```

Once an alias is defined, if you type `'name'`, the shell will automatically expand it to `'command'`. As an example, let's say you frequently run the command `'lpr -Pcolor155'`, and want to call it something shorter. To do this, you'd add

```
alias colorprint 'lpr -Pcolor155'
```

Additionally, you can use `\!*` to reference all the command line options fed to your alias. This allows you to do something like

```
alias lll 'ls -la \!* |more'
```

which defines a version of the `ls` command that uses `more` to feed a single page of output at a time (try it and see).

By default, MEnet provides users with some default aliases, these include:

Alias:	Expansion:
<code>dir</code>	<code>ls</code>
<code>cp</code>	<code>cp -i</code>
<code>mv</code>	<code>mv -i</code>
<code>l</code>	<code>ls -l</code>
<code>lc</code>	<code>ls</code>

Once an alias is defined, you can still access the original unaliased command by preceding it with a backslash, i.e. `'\cp file1 file2'`. Alternately, you could instead use the `unalias` command to remove an existing alias.

Adding Default Modules As MEnet users, you are already familiar with the `module` command necessary to run certain software packages. For example, before you can run `matlab`, you must first run

```
module load matlab
```

If you regularly use certain modules such as `tex` or `matlab` you can have modules loaded automatically by your shell by adding the appropriate `module` command to the end of your `.cshrc` file. In this example:

```
module load tex matlab
```

A Final `tsch` Customization Example Finally, I'll present an example of a `.tcshrc` file that uses a lot of the features we've talked about here, as well as some of our previous C-shell scripting knowledge:

```
# Sample .tcshrc file
```

```
# Load in the system default configuration
source /etc/Cshrc
```

```
# Add ~/scripts to the path
setenv PATH ${PATH}:~/scripts
```

```
# Environmental Variables:
```

```
# Setup gzip to always use maximum compression
setenv GZIP '-9f'
```

```
# If we're on ruby, set the printer to me10a.
# Otherwise set it to me10b
if ('hostname' == "ruby") then
    setenv PRINTER "me10a"
else
    setenv PRINTER "me10b"
endif
```

```
# I often mistype 'mail' as 'nauk'.
# Here I create an alias to handle that
alias nauk 'mail'
```

```
# Have the 'ssh' and 'rlogin' commands
# do tab complete based on hostnames
set hostnames=($LINUXMACHINES $SGIMACHINES $SUNMACHINES)
complete ssh 'p/1/$hostnames/'
complete rlogin 'p/1/$hostnames/'
```

(For information on the `complete` command, which allows you to customize the tab-completion features. Note that if you use this command you must place it in your `.tcshrc` file and *not* your `.cshrc` file, since stock `csh` doesn't understand the `complete` command).

2.1.2 `bash` Customization

`bash` is customized in a similar manner to `tcsh`. However, it uses a slightly more complicated search order. When `bash` is started as a login shell, it searches for `~/.bash_profile`, `~/.bash_login`, and `~/.profile`, in that order, executing the first one it finds that is readable. For a non-login shell, `bash` loads `~/.bashrc` instead.

In general, this means you should simply add all your customizations to `~/.bashrc`, and make sure that `~/.bash_profile` loads `~/.bashrc`. The default MEnet `~/.bashrc` is

```
if [ -f /etc/Bashrc ]; then
    source /etc/Bashrc
fi
```

```
# Add your customizations below
```

while the default `~/.bash_profile` is

```
if [ -f ~/.bashrc ]; then
    source ~/.bashrc
fi
```

Setting Environmental Variables In `bash`, environmental variables are set in a similar manner to `tcsh`. The syntax is

```
export VAR=value
```

Otherwise, the handling of environmental variables is the same as `tcsh`.

Creating Aliases Again, configuring `bash` to use aliases is similar to `tcsh`. The syntax is

```
alias name='command'
```

(Note how it only varies from `tcsh` by the `=` sign). `bash` aliases don't support `\!*` like `tcsh`, however you can achieve similar functionality by using "functions". Consult the `bash` documentation for more information.

Adding Default Modules If you regularly use certain modules such as `tex` or `matlab` you can have modules loaded automatically by your shell by adding the appropriate `module` command to the end of your `.bashrc` file:

```
module load tex matlab
```

2.1.3 If you can't log in anymore...

Since the first thing a shell does when you start it is read its own configuration files, if there is an error in your config files it can result in your not being able to log in! To avoid this problem:

1. Before making changes to your startup files, make a copy of them with a `.bak` extension. For example, `cp .cshrc .cshrc.bak`.
2. Test your changes by starting a new `xterm` or sub-shell before logging out to see if your changes work.
3. If you do get locked out of your account by an incorrect startup file, you can usually still ftp into your account and copy the original file over your incorrect copy.
4. Otherwise, come see the MEnet staff.

Also remember that copies of the default MEnet startup scripts are available in the `~newuser/` directory.

⁹Note that creating this file will cause the default SGI Desktop Environment not to be loaded

2.2 X Startup Scripts

Like your shell, the X Window System also has startup files which it loads when in log into one of the workstations through the X login screen. Depending on the operating system, the startup file has a different name:

Platform:	X Startup File:
Linux	<code>~/.xsession_linux</code>
Sun	<code>~/.xsession_sun</code>
SGI	<code>~/.xsession_sgi</code> ⁹

Each of these files is simply a shell script (so it must start with either `#!/bin/csh` or `#!/bin/bash`) that does two things:

1. Starts some basic X programs such as `xbiff`, `xclock`, `xload`, etc.
2. Starts a "window manager" (usually `fvwm2`), which allows you to interact with the windows.

The first step is optional, but the second step is mandatory. Thus a minimal `~/.xsession_*` file would have to contain

```
#!/bin/csh
exec fvwm2
```

If you were to do this and log into a machine, all you would get would be a window manager—no icons, no windows, nothing. Not very useful, so most users like to start up a few `xterms` and some other programs like `xbiff` or `xclock`.

Since the startup script for X is simply a shell script, you can do anything you normally would program into a shell script. The only difference is that you should run X programs in the background (placing an `&` after them in your script), and you should also provide a `-geometry` specifier to your program. A geometry specifier is of the form `HxW+X+Y`, where `H` and `W` are the height and width of the window, and `X` and `Y` are the horizontal and vertical location of the window. Note that either `X` or `Y` can be negative, meaning that the window be placed `X` or `Y` pixels from the right or bottom borders instead.

So, let's say that we wanted our Linux login environment to have a solid blue background (using `xsetroot`), a copy of `xbiff` running at the upper right, an `xterm` of size 80 by 25 running near the middle of the screen, and a copy of `xclock` running at the bottom right corner. The `~/.xsession_linux` file would then look like:

```
#!/bin/csh

xsetroot -solid blue
xbiff -geometry 64x64-0+0 &
xterm -geometry 80x25+100+100 &
xclock -geometry 100x100-0-0 &

exec fvwm2
```

If you mess up your X startup scripts... Like your shell startup scripts, if you mess up your X startup scripts you can easily find yourself unable to log in. Thus, you should take great care when you edit your files. However, you may still occasionally make mistakes and not be able to log in to fix them. There are two ways to handle this:

1. Find a text login console, such as telneting from a non-MEnet machine, and use your text login session to fix errors.
2. Use X's "failsafe" login: At the login screen, type your name and password, but *don't* hit return after your password. Instead hit **f1** and the computer will log you in with a (very) minimal environment, in which you can run an editor and fix your errored startup file.

Making the SGIs look like the Sun or Linux machines

For users that prefer the `fvwm2` environment of the Sun and Linux workstations to the default SGI environment, you can easily set up the SGIs to use `fvwm2`:

1. Create a `.xsession_sgi` file as discussed above. Make sure it is executable.
2. Create a file in your home directory called `.disableDesktop`, which disables the SGI desktop environment.
3. Log out and log back in.

2.3 X Defaults

Another way that many X applications can be configured is by the use of "X Applications Defaults". This allows you to tailor some aspects of X applications to your personal preferences (for example, font size, background color, etc.).

These preferences are stored in a file called `~/.Xresources`, or `~/.Xdefaults` on the Suns. Each line in this file has the form

```
Resource: value
```

Each resource is specified as hierarchical strings of the form

```
Class.[subclass.subsubclass...].option
ApplicationName.option
```

The `Class` for any application is given in the manual page for that program, but is generally the name of the application with the first one or two letters capitalized. For example, the `xterm` application's class is `'XTerm'`, `emacs`'s class is `'Emacs'`, and `xv` is of class `'XV'`.

Some applications, such as `xterm`, which have multiple display windows may have subclasses for each display window. In the case of `xterm`, the man page tells us that these subclasses are called `XTerm.vt100` and `XTerm.tek4014` for the `vt100` and `tektronics` display windows, respectively.

The available values for 'option' can be found by looking at the man pages for various X applications. For example, if we '`man xterm`', we'll see that the man page has a section entitled 'Resources' showing the available options for an application. One of these under the `XTerm.vt100` section is

```
background (class Background)
    Specifies the color to use for the background of
    the window. The default is 'white.'
```

Thus, if we added the line

```
XTerm.vt100.background: midnightblue
XTerm.vt100.foreground: grey90
```

to `~/.Xresources`.¹⁰ then your `xterm` windows would allow appear with very dark blue backgrounds and light gray letters.

Thus, if you regularly like to have `xterm` show scrollbars, appear with light gray backgrounds, a dark blue foreground, a `7x14` font, and 500 lines of scrollbar, from the `xterm` man page we can see that this would be accomplished with the following lines in `~/.Xresources`:

```
XTerm*background:          grey90
XTerm.vt100.background:   grey90
XTerm.vt100.foreground:   MidnightBlue
XTerm.vt100.cursorColor:  Black
XTerm*font:               7x14
XTerm*boldFont:           7x14bold
XTerm.vt100.scrollBar:    True
XTerm*saveLines:          500
```

This is only a brief introduction to X resources and their capabilities. For more information see the X documentation ('`man X`') and individual application documentation.

2.4 Other Account Configuration

There are a number of additional ways you can configure your account. I'll briefly mention each of these:

2.4.1 fvwm2

`fvwm2`, the window manager used by default on the Sun and Linux workstations, is highly configurable. `fvwm2` is configured by making changes to your `~/.fvwm2rc` file in your home directory¹¹ By making changes to this file, you can change the fonts, colors, menus, and other `fvwm2` features. If you examine this file, you'll note that it's fairly well documented, with most of the lines including comments on their functions.

For example, the default `~/.fvwm2rc` file contains these entries which generate the "applications" menu accessed from the background window:

¹⁰To make changes to your `.Xresources` file take effect, you must log out and log back in, or run `'xrdp ~/.Xresources'`

¹¹Like always, if this file is broken or missing, you can obtain a replacement from the `newuser` account.

```
AddToMenu "Applications" "Applications" Title
+ "%mini.xterm.xpm%xterm" Exec exec xterm &
+ "" Nop
+ "%mini.emacs.xpm%Emacs" Exec exec emacs &
+ "%mini.netscape.xpm%Netscape" Exec exec netscape &
```

If we wanted to add another program, such as `xv` to this menu, it is simply a matter of extending the menu definition above and restarting `fvwm2`:

```
AddToMenu "Applications" "Applications" Title
+ "%mini.xterm.xpm%xterm" Exec exec xterm &
+ "" Nop
+ "%mini.emacs.xpm%Emacs" Exec exec emacs &
+ "%mini.netscape.xpm%Netscape" Exec exec netscape &
+ "XV" Exec exec xv &
```

For more information on customization your `~/fvwm2rc`, please consult the `fvwm2` man page.

Like any other account customization, if you make changes to this file please save the original version, since if you make incorrect changes you may have troubles logging in (see the previous section on “fail-safe” X logins).

3 Useful Programs

3.1 Make

Many MEnet users use the workstations to write and develop programs in C and Fortran.

`make` helps automate the editing/compiling/executing process by introducing the concept of *dependencies*. For example, let's say that I have a program `dvm2` which is compiled from the programs `dvm2.c` and `four1.c`, and needs to be compiled with the command

```
gcc -o dvm2 dvm2.c four1.c /usr/local/lib/gpib/cib.o -lm
```

Typing this command every time I change either `dvm2.c` or `four1.c` becomes rather tedious. The purpose of the `make` utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them. This is accomplished by creating an input file called a `Makefile`.

A `makefile` contains “rules” which tell `make` what “targets” exist (usually a target is a desired executable), what those targets depend on, and how to generate the target from the dependencies. A typical rule looks like

```
target: prerequisites
      commands
```

(Note that in `Makefiles` the lines after the target in a `makefile` *must* begin with a tab and not spaces). `Makefiles` can also contain macro definitions of the form

```
MACRO = value
```

and then the value of `MACRO` can be accessed in rules files as `$(MACRO)`. Often it is useful to set things like the name of the compiler and a list of flags to feed to the compiler. So, as an example for the above program, one could write a `Makefile` like:

```
CC = gcc
LFLAGS = /usr/local/lib/gpib/cib.o -lm

dvm2: dvm2.c four1.c
      $(CC) -o dvm2 dvm2.c four1.c $(LFLAGS)
```

Now any time I want to make sure that I have an up-to-date version of `dvm2`, all I have to do is type ‘`make dvm2`’¹² and if any changes have been made to either `dvm2.c` or `four1.c`, `make` will recompile `dvm2`.

Additionally, you can include multiple targets in a `makefile`. Also, if you tell `make` that a file depends on a `.o` file and don't give any rules to generate those files from any `.c` files, it assumes you want to compile them with the rule¹³:

```
$(CC) -c file.c -o file.o
```

So, a more complicated example that compiles two different programs, `dvm` and `dvm2`, could look like

```
CC = gcc
LFLAGS = /usr/local/lib/gpib/cib.o -lm

all: dvm dvm2

dvm: dvm.o
      $(CC) -o dvm dvm.o $(LFLAGS)

dvm2: dvm2.o four1.o
      $(CC) -o dvm2 dvm2.o four1.o $(LFLAGS)
```

See the `make` documentation for more details, or see chapter 19 in *Unix in a Nutshell*.

3.2 Diff/Patch

`diff` is a Unix utility that lists the differences (or “diffs”) between two text files. Usually it is used to compare two different versions of the same file (like if you've made changes to a `.c` file you're working on). The usage is

```
diff [options] from-file to-file
```

The best way to show `diff` is with an example. Let's say I have a program, `hello.c`:

¹²`make` with no arguments will choose the first target in the `makefile` unless a `default` target is defined

¹³`make` has a lot of implied rules like this, see the `make` documentation for more info

```
#include<stdio.h>

main ()
{
    printf("Hello, World!\n");
}
```

and then we edit this program to change it into `hello2.c`:

```
#include<stdio.h>

main ()
{
    printf("Bonjour, Le Monde!\n");
}
```

`diff` can give its output in a number of different formats, but the *unified* output (the `-u`) option is generally the most useful, see the documentation for other formats). If we run `'diff -u hello.c hello2.c` on them we get:

```
--- hello.c      Thu Aug 13 14:38:01 1998
+++ hello2.c     Thu Aug 13 14:44:19 1998
@@ -2,5 +2,5 @@

main ()
{
-   printf("Hello, World!\n");
+   printf("Bonjour, Le Monde!\n");
}
```

We can see from this output first that it is comparing files `hello.c` and `hello2.c`, with their dates. Then after each `@@` it lists the portions of the files where they differ, with a few lines of context. Here it shows that the line starting with `-` has been removed, and the line marked `+` has been added.

Where this really becomes useful is with the partner program to `diff`, called `patch`. `patch` takes the output of `diff` (usually stored in a file ending in `.patch`, called a “patch file”), and uses it to generate a new version of a file from the old version and the patch file.

For example, let’s say me and a friend are both working on a large program `bigprog.c`, and I’ve made a number of small changes to `bigprog.c` to make `bigprog2.c`. My friend already has `bigprog.c`, so instead of sending him the entirety of `bigprog2.c`, I just have to send him the output of

```
diff -u bigprog.c bigprog2.c > bigprog.patch
```

(which is almost certainly smaller than `bigprog2.c`) and then my friend can create his own copy of the latest `bigprog` by simply running

```
patch < bigprog.patch
```

For more information on `diff` and `patch`, read their man pages.

3.3 ssh

`ssh` (“Secure Shell”) is a replacement for `rlogin` and `telnet` that provides secure encrypted communications between two machines¹⁴. `ssh` is available on all MEnet, AEM, MSI, and ITlabs workstations.

Basically, `ssh` is used in the same manner as `telnet` or `rlogin`,

```
ssh [options] hostname
```

In addition to providing secure communications, `ssh` has some additional side-benefits:

1. `ssh` automatically handles the setting of `DISPLAY` environmental variable, so if you log into another machine and run graphical programs they should automatically show up on your local machine.
2. If invoked with `ssh -C` then `ssh` will also compress the data transported between machines. This is very useful if you are working over a slow connection like a modem line.
3. If you use a `ssh` with “authentication keys”, you can set up automatic authentication such that when you log in, you enter a pass-phrase once and then you can access all your remote account without a password (this is like the old `.rhosts` files, but much more secure).

MEnet users are encouraged to use `ssh` instead of `telnet` or `rlogin`. Further information on using `ssh` can be found on the web, <http://www.tac.nyc.ny.us/~kim/ssh/>, and by consulting the `ssh` man page.

3.4 Other Useful Programs

There are a number of other useful Unix programs that you should be aware of, although most of these programs are too detailed to cover in a short course. For most of these I’ll just give a sample of what they are capable of, and where you can look for more information.

3.4.1 awk

`awk` is a scripting language that processes files using pattern-matching (using patterns called “Regular Expressions”). Basically, `awk` reads in an input file, separates each input line into “fields” (usually fields are delimited by whitespace), and runs a series of user-specified commands on each input line. The best way to show the sorts of things that `awk` can do is with an example.

This example calculates the total number of bytes used by the files in a directory¹⁵, by examining the output of `'ls -lg'` (the fifth column of the output is the file size, hence the script uses `5$`):

¹⁴This is useful because many times MEnet systems have been compromised by passwords obtain from unsecure communications

¹⁵Note that the standard Unix `du` utility a better way of accomplishing this

```
ls -lg FILES | awk '{ x += $5 } ; END { print "total bytes: " x }'
```

For more information on `awk`, consult the `awk` man page, the `awk` info page, the `awk` chapter in *Unix in a Nutshell*, or find a copy of O'Reilly and Associates *Sed & Awk* manual.

3.4.2 sed

`sed`, the “Stream Editor”, is similar to `awk` in that it is a utility that reads in a file and applies commands to each line. Unlike `awk`, `sed` is designed to apply editing commands to files, such as substituting all instances of one string with another. For example, if I wanted to replace all instances of my old home directory, `/home/home9/kaszeta` with my new home directory, `/home/home18/kaszeta` in a given file, the command would be:

```
sed 's/home9/home18/g' oldfile > newfile
```

For more information on `sed`, consult the `sed` man page, the `sed` chapter in *Unix in a Nutshell*, or find a copy of O'Reilly and Associates *Sed & Awk* manual.

3.4.3 perl

`perl` is the “Practical Extraction and Report Language”, a scripting language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. `perl` is intended to be practical and easy to use. `perl` is also commonly used to create CGI scripts for dynamic web sites.

`perl`'s scripting language is basically a combination of the features of `C`, `sed`, `awk`, and shell scripts, so that a user that is familiar with any of these should find `perl` easy to learn.

As an example of what a typical `perl` script looks like, this script processes the output of `gfinger` to count the number of unique users logged into all MEnet workstations¹⁶:

```
#!/usr/local/bin/perl
#
# Counts the number of users
# currently logged in to the system
#
# Spawn a gfinger to obtain raw data from
open(GFINGER, "/usr/local/bin/gfinger|");

# Start with a zero count
$count=0; $lastuser="";

# Throw away first line
$junk=<GFINGER>;

while ($fingerline = <GFINGER>) {
    if($fingerline !~ /root/i) {
        if($fingerline !~ /ftp/i) {
            if($fingerline !~ /admin/i) {
                $fingerline=~s/\W.*//;
```

```
                chop($fingerline);
                if($fingerline ne $lastuser) {
                    $lastuser=$fingerline;
                    $count++;}}}}
    }
$time=time();

# Dump the data to the datafile;
print("$count\n");
```

For more information on learning `perl`, the best place to start is *Learning Perl*, from O'Reilly and Associates. Once you've learned basic `perl`, another good resource is *Programming Perl*, also from O'Reilly and Associates.

4 Unix Tips and Tricks

4.1 xterm Tricks

- First of all, many people are unaware that `xterm` has menus. You can access these menus by holding down the `ctrl` key and clicking on the window with any of the mouse buttons. The left and middle mouse button brings up lists of options such as toggling scroll bars, resetting the terminal and such. The right mouse button brings up a menu of differently sized fonts.
- If you don't want to use scrollbars in `xterm`, you can still scroll up and down in the window by using `shift-pgup` and `shift-pgdn`.
- Also, there are a lot of useful mouse-clicking commands. You can select text with the left mouse button, and paste with the middle button. The right hand button extends an existing selection. Also note that double-clicking will select an entire word, and triple-clicking selects an entire line.
- Finally, how `xterm` determines what constitutes a “word” is defineable through X resources. By default, `xterm` defines a “word” as any group of letters separated by non-letters. However, it's possible to make `xterm` treat exclamation marks, percent signs, dashes, slashes, and ampersands as parts of words as well. To do this, add this to your `~/.Xresources`¹⁷:

```
XTerm*charClass: 33:48,37:48,45-47:48,64:48
```

4.2 Emacs

The Emacs editor is also very configurable. Some useful configurations that you make not be aware of:

- Emacs supports *color syntax highlighting*, in which portions of the edited text are colorized according to their type. For example, in `C` programs, the keywords, quotations, and data types are highlighted to make them easier to pick out. You can enable this feature by adding the following to `~/.emacs`:

¹⁶A variation of this script is used to generate the MEnet usage graphs, url—<http://www.menet.umn.edu/stats/mrtg-2.5.2/web/user-count.html>—.

¹⁷To further customize this, or understand how this works, see the `xterm` man page

```
;; Turn on font-lock mode
(global-font-lock-mode 1 t)
```

- If you regularly view or edit documents with accented characters such as á or ñ in them, emacs usually presents these as sequences like \341 or \361. To have them display correctly, add the following to ~/.emacs:

```
;; Fix accented chars
(standard-display-european t)
```

- Whenever you type or select a delimiting character such as (or] in emacs, it normal will show you by jumping the cursor where the matching delimiter is. You can have it instead show the matching delimiter by color highlighting using

```
;; Better parentheses matching
(load-library "paren")
```

- If you've seen it enough times, you can get rid of the emacs startup screen using

```
;;Get rid of the startup message
(setq inhibit-startup-message t)
```

- Emacs likes to start up in fundamental mode or lisp mode. To have it start in text mode, simply add

```
;; Make emacs default to text mode
(setq default-major-mode 'text-mode)
```

5 Pointers to further information

For more information, there are a number of good Unix resources on the web, including:

- <http://www.geek-girl.com/unix.html>, "The Unix Reference Desk".
- <http://www.linuxbox.com/~taylor/4ltrwrdr/>, "Unix is a 4 Letter Word".
- <http://www.cis.ohio-state.edu/hypertext/faq/usenet/unix-faq/faq/top.html>, "The Unix Frequently Asked Questions List".

Otherwise, feel free to ask myself or the MEnet staff in room 155.