Shells

- Shells are command interpreters
  - they allow interactive users to execute the commands.
  - typically a command causes another program to be run
- shells may have a graphical (point-and-click) interface
  - e.g. Windows or Mac desktop
  - much easier for naive users
  - much less powerful & not covered in this course
- command-line shells are programmable, powerful tools for expert users
- **`bash`** is the most popular used shell for unix-like systems
- other significant unix-like shells include : **`dash`**, **`zsh`**, **`busybox`**
- we will cover the core features provided by all shells
  - essentially the POSIX standard shell features

What Shells Do

- Unix shells have the same basic mode of operation:

```
loop
   if (interactive) print a prompt
   read a line of user input
   apply transformations to line
   split line into words using whitespace
   use first word in line as command name
   execute command, passing other words as arguments
end loop
```

- shells can also be run with commands in a file
- shells are programming languages
- shells have design decisions to suit interactive use
  - e.g. variables don't have to be initialized or declared
  - these decisions not ideal for programming in Shell
  - in other words there have to be design compromises

Processing a Shell Input Line

- a series of **transformations** are applied to Shell input lines
  1. variable expansion, e.g. `$HOME` → `/home/z1234567`
  2. command expansion e.g. **`$(whoami)`** → `z1234567`
  3. arithmetic, e.g. `$((6 * 7))` → **42**
  4. word splitting - line is broken up on white-space unless inside quotes
  5. pathname globbing, e.g. `*.c` → `main.c i.c`
  6. I/O redirection e.g. `<i.txt` → stdin replaced with stream from `i.txt`
  7. first word used as program name, other words passed as arguments
- order of these transformation is important!
- not understanding order is a common source of bugs & security holes
  - shell is better-avoided if security is significant concern
- directories in `PATH` searched for program name

`echo` - print arguments to stdout

- **`echo`** prints its arguments to stdout
- mainly used in scripts, but also useful when exploring shell behaviour
- **`echo`** is often builtin to shells for efficiency, but also provided by **`/bin/echo`**
- see also **`/usr/bin/printf`** - not POSIX but widely available
- Two useful **`echo`** options:

-n	do not output a trailing newline
-e	enable interpretation of backslash escapes

```
$ echo Hello Andrew
Hello Andrew
$ echo '\n'
\n
$ echo -e '\n'


$ echo -n Hello Andrew
Hello Andrew$
```

echo in Python:

```python
import sys
def main():
    """
    print arguments to stdout
    """
    print(' '.join(sys.argv[1:]))
```

source code for echo.py

echo in C:

```c
int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        if (i > 1) {
            fputc(' ', stdout);
        }
        fputs(argv[i], stdout);
    }
    fputc('\n', stdout);
    return 0;
}
```

source code for echo.c

Shell Variables

- shell variables are untyped - consider them as strings
  - note that 1 is equivalent to **"1"**
- shell variables are not declared
- shell variables do not need initialization
  - initial value is the empty string
- one scope - no local variables
  - except sub-shells & functions (sort-of)
  - changes to variables in sub-shells have no effect outside sub-shell
  - components of pipeline executed in sub-shell
- `$name` replaced with value of variable `name`
- `name=value` assigns `value` to variable `name`
  - note: no spaces around =

`$(command)` - command expansion:

- `$(`***command***`)` is evaluated by running ***command***
- stdout is captured from ***command***
- `$(`***command***`)` is replaced with the entire captured stdout
- **'*command*'** (backticks) is equivalent to `$(`***command***`)`
  - backticks is original syntax, so widely used
  - nesting of backticks is problematic

For example:

```
$ now=$(date)
$ echo $now
Sun 23 Jun 1912 02:31:00 GMT
$
```

'' - Single Quotes

- single quotes `''` group the characters within into a single word
  - no characters interpreted specially inside single quotes
  - a single quote can not occur within single quotes
  - you can put a double quote between single-quotes

For example:

```
$ echo '***   !@#$%^&*(){}[]:;"<>?,./`   ***'
***   !@#$%^&*(){}[]:;"<>?,./`   ***
$ echo 'this is "normal"'
this is "normal"
```


"" - Double Quotes

- double quotes `""` group the characters within into a single word
  - variables and commands are expanded inside double-quotes
  - backslash can be used to escape $ “ "" ” \
  - other characters not interpreted specially inside double quotes
  - you can put a single quote between double-quotes

For example:

```
$ answer=42
$ echo "The answer is $answer."
The answer is 42.
$ echo 'The answer is $answer.'
The answer is $answer.
$ echo "time's up"
time's up
```


<< - here documents

- <<***word*** called a here document
- following lines until ***word*** specify multi-line string as command input
- variables and commands expanded - same as double quotes
- <<‘***word***’ variables and commands not expanded - same as single quotes

```
$ name=Andrew
$ tr a-z A-Z <<END-MARKER
Hello $name
How are you
Good bye
END-MARKER
HELLO ANDREW
HOW ARE YOU
GOOD BYE
```


Arithmetic

- `$((`***expression***`))` is evaluated as an arithmetic expression
- ***expression*** is evaluated using C-like integer arithmetic
- `$((`***expression***`))` is replaced with the result
- the `$` on variables can be omitted in expression (must an contain integer)
- shell arithmetic implementation slow compared to e.g. C
  - significant overhead converting to/from strings
- older scripts may use the separate program **`expr`** for arithmetic

For example:

```
$ x=8
$ answer=$((x*x - 3*x + 2))
$ echo $answer
42
```

word splitting

- coders not understanding how shells split words is a frequent source of bugs

```
# inspect how shell splits lines into program arguments (argv)
import sys
print(f'sys.argv = {sys.argv}')
```

source code for print_argv.py

```
$ v=''
$ ./print_argv.py $v
sys.argv = ['./print_argv.py']
$ ./print_argv.py "$v"
sys.argv = ['./print_argv.py', '']
$ w='  xx    yyy        zzzz   '
$ ./print_argv.py $w
sys.argv = ['./print_argv.py', 'xx', 'yyy', 'zzzz']
$  ./print_argv.py "$w"
sys.argv = ['./print_argv.py', '  xx    yyy        zzzz   ']
```


*?[]! - pathname globbing

- `*?[]!` characters cause a word to be matched against pathnames
  - confusingly similar to regexes - but much less powerful
- `*` matches 0 or more of **any** character - equivalent to regex `.*`
- `?` matches any **one** characters - equivalent to regex `.`
- `[`***characters***`]` matches **1** of ***characters*** - same as regex `[]`
- `[!`***characters***`]` matches **1** character not in ***characters*** - same as regex `[^]`
- if no pathname matches the word is unchanged
- aside: globbing also available in Python, Perl, C & other languages

```
$ echo *.[ch]
functions.c functions.h i.h main.c
$ ./print_argv.py  *.[ch]
['./print_argv.py', 'functions.c', 'functions.h', 'i.h', 'main.c']
$ ./print_argv.py  '*.[ch]'
['./print_argv.py', '*.[ch]']
$ ./print_argv.py  "*.[ch]"
['./print_argv.py', '*.[ch]']
$ ./print_argv.py *.zzzzz
['./print_argv.py', '*.zzzzz']
```


I/O Redirection

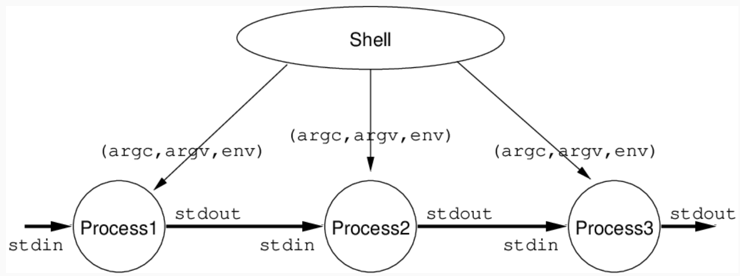
- stdin, stdout & stderr for a command can be directed to/from files

< ***infile***	connect `stdin` to the file `infile`
> ***outfile***	send `stdout` to the file `outfile`
>> ***outfile***	append `stdout` to the file `outfile`
2> ***outfile***	send `stderr` to the file `outfile`
2>> ***outfile***	append `stderr` to the file `outfile`
> ***outfile*** 2>&1	send `stderr`+`stdout` to `outfile`
1>&2	send `stdout` to `stderr` (handy for error messages)

- beware: > truncates file before executing command.
- always have backups!

Pipelines

- ***command*** $_1$ | ***command*** $_2$ | ***command*** $_3$ | ...
- stdout of ***command*** $_{n-1}$ connected to stdin of ***command*** $_n$
- beware changes to variables in pipeline are lost
- some non-filter style Unix programs given a filename − read from stdin
  - allows them to be used in a pipeline



searching PATH for the program

- first word on line specifies command to be run
- if first word is not the full (absolute) pathname of a file the colon-separated list of directory specified by the variable PATH is searched
- for example if **PATH=/bin/:/usr/bin/:/home/z1234567/bin** and the command is **kitten** the shell will check (stat) these files in order:
  - **/bin/kitten /usr/bin/kitten /home/z1234567/bin**
  - the first that exists and is executable will be run
  - if none exist the shell will print an error message
- or . in PATH causes the current directory to be checked
  - this can be convenient - but make it last not first, e.g.: **PATH=/bin/:/usr/bin/:/home/z1234567/bin:.**
  - definitely do not include the current directory in PATH if you are root
  - an empty entry in PATH is equivalent to .

danger of having . in your PATH

- if . is not last in PATH then programs in the current directory may be unexpectedly run
- this can also happen inside run shell scripts or other programs you run
- robust shell scripts often set PATH to ensure this doesn't happen, e.g.: **PATH=/bin/:/usr/bin/:$PATH**

```
# equivalent to PATH=.:/bin:/usr/bin:/home/z1234567/bin
$ PATH=:/bin:/usr/bin:/home/z1234567/bin
$ cat >cat <<eof
#!/bin/sh
echo miaou
eof
$ chmod 755 cat
$ cat /home/cs2041/public_html/index.html
miaou
$
```

Problem: ./cat is being run rather /bin/cat

Shell Scripts

We can execute shell commands in a file:

```
$ cat hello
echo Hello, John Connor - the time is $(date)
$ sh hello
Hello, John Connor - the time is Fri 29 Aug 1997 02:14:00 EST
```

- Unix-like systems allow an interpreter to be specified in a #! line
- allows program to be executed directly without knowing it is shell

```
$ cat hello
#!/bin/sh
echo Hello, John Connor
echo The time is $(date)
$ chmod 755 hello
$ ./hello
Hello, John Connor - the time is Fri 29 Aug 1997 02:14:00 EST
```

- use **#!/bin/bash** if you want bash

Shell Built-in Variables

Some shell built-in variables with pre-assigned values:

`$0`	the name of the command
`$1`	the first command-line argument
`$2`	the second command-line argument
...	...
`$9`	the ninth command-line argument
`${10}`	the tenth command-line argument
...	...
`${255}`	the two hundred and fifty-fifth (last) command-line argument
`$#`	count of command-line arguments
`$*`	all the command-line arguments (separately)
`"$*"`	all the command-line arguments (together)
`$@`	all the command-line arguments (separately)
`"$@"`	all the command-line arguments (as quoted)
`$?`	exit status of the most recent command
`$$`	process ID of this shell

Example - Shell Script using Built-in Variables

```
#!/bin/dash
# A simple shell script demonstrating access to arguments.
# written by andrewt@unsw.edu.au as a COMP(2041|9044) example
echo My name is "$0"
echo My process number is $$
echo I have $# arguments
# your not going to see any difference unless you use these in a lo
echo My arguments separately are $*
echo My arguments together are "$*"
echo My arguments separately are $@
echo My arguments as quoted are "$@"
echo My 5th argument is "'$5'"
echo My 10th argument is "'${10}'"
echo My 255th argument is "'${255}'"
```

source code for args.sh

Example - Simple Shell Script

```
#!/bin/sh
# l [file|directories...] - list files
#
# written by andrewt@unsw.edu.au as a COMP(2041|9044) example
#
# Short shell scripts can be used for convenience.
#
# It is common to put these scripts in a directory
# such as /home/z1234567/scripts
# then add this directory to PATH e.g in .bash_login
# PATH=$PATH:/home/z1234567/scripts
#
# Note: "$@" like $* expands to the arguments to the script,
# but preserves whitespace in arguments.
ls -las "$@"
```

source code for l

Example - Putting a Pipeline in a Shell Script

```
#!/bin/sh
# Count the number of time each different word occurs
# in the files given as arguments, or stdin if no arguments,
# e.g. word_frequency.sh dracula.txt
# written by andrewt@unsw.edu.au as a COMP(2041|9044) example
cat "$@" |                   # tr doesn't take filenames as argumen
tr '[:upper:]' '[:lower:]' | # map uppercase to lower case
tr ' ' '\n' |                # convert to one word per line
tr -cd "a-z'" |              # remove all characters except a-z anc
grep -E -v '^$' |            # remove empty lines
sort |                       # place words in alphabetical order
uniq -c |                    # count how many times each word occur
sort -rn                     # order in reverse frequency of occurr
# notes:
# - first 2 tr commands could be combined
# - sed 's/ /\n/g' could be used instead of tr ' ' '\n'
# - sed "s/[^a-z']//g" could be used instead of tr -cd "a-z'"
```

source code for word_frequency.sh

Exit Status and Control

- when Unix-like programs finish they give the operating system an **exit status**
  - the return value of ‘main becomes the **exit status** of a C program
  - or if exit is called, its argument is the **exit status**
  - in Python **exit status** is supplied as an argument to `sys.exit`
- an **exit status** is a (usually small) integer
  - by convention a zero exit status indicated normal/successful execution
  - a non-zero exit status indicates an error occurred
  - which non-zero integer might indicate the nature of the problem
- program exit status is often ignored
  - not important writing single programs (COMP1511/COMP9021)
  - very important when combining multiple programs (COMP2041/COMP9044)
- flow of execution in Shell scripts based on exit status
  - `if`/`while` statement conditions use exit status
- two weird utilities
  - `/bin/true` does nothing and always exits with status 0
  - `/bin/false` does nothing and always exits with status 1

The `test` command

- The **`test`** command performs a test or combination of tests and:
  - does/prints nothing
  - returns a zero exit status if the test succeeds
  - returns a non-zero exit status if the test fails
- Provides a variety of useful operators:
  - string comparison: `= !=`
  - numeric comparison: `-eq -ne -lt`
  - test if file exists/is executable/is readable: `-f -x -r`
  - boolean operators (and/or/not): `-a -o !`
- also available as '[' instead of test - which many programmers prefer
- builtin to some shell (e.g. bash) but available as **`/bin/test`** or **`/bin/[`**

The `test` command examples

```
# does the variable msg have the value "Hello"?
test "$msg" = "Hello"

# does x contain a numeric value larger than y?
test "$x" -gt "$y"

# Error: expands to "test hello there = Hello"?
msg="hello there"
test $msg = Hello

# is the value of x in range 10..20?
test "$x" -ge 10 -a "$x" -le 20

# is the file xyz a readable directory?
test -r xyz -a -d xyz

# alternative syntax; requires closing ]
[ -r xyz -a -d xyz ]
```


Using Exit Status for Conditional Execution

- all commands are executed if separated by **`;`** or newline, e.g:
  cmd$_1$ `;` *cmd*$_2$ `;` ... `;` *cmd*$_n$
- when commands are separated by **`&&`**
  cmd$_1$ `&&` *cmd*$_2$ `&&` ... `&&` *cmd*$_n$
  execution stops if a command has non-zero exit status
  cmd$_{n+1}$ is executed only if *cmd*$_n$ has zero exit status
- when commands are separated by `||`
  cmd$_1$ `||` *cmd*$_2$ `||` ... `||` *cmd*$_n$
  execution stops if a command haszero exit status
  cmd$_{n+1}$ is executed only if *cmd*$_n$ has non-zero exit status
- `{}` can be used to group commands
- `()` also can be used to group commands - but executes them in a subshell
  - changes to variables and current working directory have no effect outside the subshell
- exit status of group or pipeline of commands is the last exit status

Conditional Execution Examples

```
# run a.out if it exists and is executablr
test -x a.out && ./a.out

# if directory tmp doesn't exist create it
test -d tmp || mkdir tmp

# if directory tmp doesn't exist create it
{test -d tmp || mkdir tmp;} && chmod 755 tmp
# but simpler is
mkdir -p tmp && chmod 755 tmp
```

{} versus () - example

```
$ cd /usr/share
$ x=123
$ ( cd /tmp;  x=abc; )
$ echo $x
123
$ pwd
/usr/share
$ { cd /tmp;  x=abc; }
$ echo $x
abd
$ pwd
/tmp
```

- changes to variables and current working directory have no effect outside a subshell
- pipelines also executed in subshell, but variables and directory not usually changed in a pipeline

If Statements in Shell

- shell if statements have this form:

`if` *command$_1$*
`then`
    then-commands
`elif` *command$_2$*
`then`
    elif-commands
`else`
    else-commands
`fi`

- the execution path depends on the exit status of *command$_1$* and *command$_2$*
- ***command$_1$*** is executed and if its exit status is 0,
  the ***then-commands*** are executed
- otherwise ***command$_2$*** is executed and if its exit status is 0,
  the ***elif-commands*** are executed

If Statements - Example

```
if gcc main.c
then
    echo your C compiles
elif python3 main.c
    echo you have written Python not C
else
    echo program broken - send help
fi
```

```
if gcc a.c
then
    # you can not have an empty body
    # use a : statement which does nothing
    :
else
    rm a.c
fi
```


While Statements in Shell

- shell while statements have this form:

`while` *command*
`do`
    body-commands
`done`

- the execution path depends on the exit status of *command*
- ***command*** is executed and if its exit status is 0,
  the ***body-commands*** are executed
  and then ***command*** is executed and if its exit status is 0
  the ***body-commands*** are executed
  and . . .
- if the exit status of ***command***~ is not 0, execution of the loop stops

example - seq - simple version

```
#!/bin/sh
# simple emulation of /usr/bin/seq for a COMP(2041|9044) example
# andrewt@unsw.edu.au
# Print the integers 1..n with no argument checking
last=$1
number=1
while test $number -le "$last"
do
    echo $number
    number=$((number + 1))
done
```

source code for seq.v0.sh

```
$ ./seq.v0.sh 3
1
2
3
```


example - seq - argument handling added

```
# Print the integers 1..n or n..m
if test $# = 1
then
    first=1
    last=$1
elif test $# = 1
then
    first=$1
    last=$2
else
    echo "Usage: $0 <last> or  $0 <first> <last>" 1>&2
fi
number=$first
while test $number -le "$last"
do
    echo $number
    number=$((number + 1))
done
```

source code for seq.v1.sh

example - seq - using [] instead of test

```
if [ $# = 1 ]
then
    first=1
    last=$1
elif [ $# = 1 ]
then
    first=$1
    last=$2
else
    echo "Usage: $0 <last> or  $0 <first> <last>" 1>&2
fi
number=$first
while [ $number -le $last ]
do
    echo $number
    number=$((number + 1))
done
```

source code for seq.v2.sh

example - seq - using [] instead of test

```
if [ $# = 1 ]
then
    first=1
    last=$1
elif [ $# = 1 ]
then
    first=$1
    last=$2
else
    echo "Usage: $0 <last> or  $0 <first> <last>" 1>&2
fi
number=$first
while [ $number -le $last ]
do
    echo $number
    number=$((number + 1))
done
```

source code for seq.v2.sh

example - watching a website - argument checking

```
# Repeatedly download a specified web page
# until a specified regexp matches its source
# then notify the specified email address.
#
# For example:
# watch_website.sh http://ticketek.com.au/ 'Ke[sS$]+ha' andrewt@uns
repeat_seconds=300  #check every 5 minutes
if test $# = 3
then
    url=$1
    regexp=$2
    email_address=$3
else
    echo "Usage: $0 <url> <regex>" 1>&2
    exit 1
fi
```

source code for watch_website.sh

example - watching a website - main loop

```
while true
do
    if curl --silent "$url"|grep -E "$regexp" >/dev/null
    then
        echo "Generated by $0" |
        mail -s "$url now matches $regexp" "$email_address"
        exit 0
    fi
    sleep $repeat_seconds
done
```

source code for watch_website.sh

For Statements in Shell

- shell for statements have this form:

for *var* **in** $word_1$ $word_3$ ... $word_n$
do
    body-commands
done

- the loop executes once for each *word* with *var* set to the *word*
- **break** & **continue** statements can be in used inside for & while loops with the same effect as C/Python
- keywords such **for**, **if**, ***while**, ... are only recognised at the start of a command, e.g.:

```
$ echo when if else for
when if else for
```


example - renaming files - argument checking

```
# Change the names of the specified files to lower case.
# (simple version of the perl utility rename)
#
# Note use of test to check if the new filename is unchanged.
#
# Note the double quotes around $filename so filenames
# containing spaces are not broken into multiple words
# Note the use of mv -- to stop mv interpreting a
# filename beginning with - as an option
# Note files named -n or -e still break the script
# because echo will treat them as an option,
if test $# = 0
then
    echo "Usage $0: <files>" 1>&2
    exit 1
fi
```

source code for tolower.sh

example - renaming files- main loop

```
for filename in "$@"
do
    new_filename=$(echo "$filename" | tr '[:upper:]' '[:lower:]')
    test "$filename" = "$new_filename" &&
        continue
    if test -r "$new_filename"
    then
        echo "$0: $new_filename exists" 1>&2
    elif test -e "$filename"
    then
        mv -- "$filename" "$new_filename"
    else
        echo "$0: $filename not found" 1>&2
    fi
done
```

source code for tolower.sh

creating a 1001 file C program - getting stared

```
# this programs create 1000 files f0.c .. f999.c
# file f$i.c contains function f$i which returns $i
# for example file42.c contains function f42 which returns 42
# main.c is created with code to call all 1000 functions
# and print the sum of their return values
#
# first add the initial lines to main.c
# note the use of quotes on eof to disable variable interpolation
# in the here document
cat >main.c <<'eof'
#include <stdio.h>
int main(void) {
    int v = 0 ;
eof
```

source code for create_1001_file_C_program.sh

creating a 1001 file C program - creating the files

```
i=0
while test $i -lt 1000
do
    # add a line to main.c to call the function f$i
    cat >>main.c <<eof
    int f$i(void);
    v += f$i();
eof
    # create file$i.c containing function f$i
    cat >file$i.c <<eof
int f$i(void) {
    return $i;
}
eof
    i=$((i + 1))
done
```

source code for create_1001_file_C_program.sh

creating a 1001 file C program - compiling & running the program

```
cat >>main.c <<'eof'
    printf("%d\n", v);
    return 0;
}
eof
# compile and run the 1001 C files
# time clang main.c file*.c
# ./a.out
```

source code for create_1001_file_C_program.sh

example plagiarism detection - simple diff

```
# written by andrewt@unsw.edu.au for COMP(2041|9044)
#
# Run as plagiarism_detection.simple_diff.sh <files>
# Report if any of the files are copies of each other
#
# Note use of diff -iw so changes in white-space or case
# are ignored
for file1 in "$@"
do
    for file2 in "$@"
    do
        test "$file1" = "$file2" &&
            break # avoid comparing pairs of assignments twice
        if diff -i -w "$file1" "$file2" >/dev/null
        then
            echo "$file1 is a copy of $file2"
        fi
    done
```


plagiarism detection - ignoring changes to comments

```
# This means changes in comments won't affect comparisons.
# Note use of temporary files
TMP_FILE1=/tmp/plagiarism_tmp1$$
TMP_FILE2=/tmp/plagiarism_tmp2$$
for file1 in "$@"
do
    for file2 in "$@"
    do
        test "$file1" = "$file2" &&
            break # avoid comparing pairs of assignments twice
        sed 's/\/\/.*//' "$file1" >$TMP_FILE1
        sed 's/\/\/.*//' "$file2" >$TMP_FILE2
        if diff -i -w $TMP_FILE1 $TMP_FILE2 >/dev/null
        then
            echo "$file1 is a copy of $file2"
        fi
    done
done
rm -f $TMP_FILE1 $TMP_FILE2
```

source code for plagiarism_detection.comments.sh

plagiarism detection - ignoring changes to variable names

```
# and change all identifiers to the letter 'v'.
# Hence changes in strings & identifiers will be ignored.
TMP_FILE1=/tmp/plagiarism_tmp1$$
TMP_FILE2=/tmp/plagiarism_tmp2$$
# s/"["]*"/s/g changes strings to the letter 's'
# It won't match a few C strings which is OK for our purposes
# s/[a-zA-Z_][a-zA-Z0-9_]*/v/g changes variable names to 'v'
# It will also change function names, keywords etc.
# which is OK for our purposes.
substitutions='
    s/\/\/.*//
    s/"[^"]"/s/g
    s/[a-zA-Z_][a-zA-Z0-9_]*/v/g'
```

source code for plagiarism_detection.identifiers.sh

plagiarism detection - ignoring changes to variable names

```
for file1 in "$@"
do
    for file2 in "$@"
    do
        test "$file1" = "$file2" &&
            break # avoid comparing pairs of assignments twice
        sed "$substitutions" "$file1" >$TMP_FILE1
        sed "$substitutions" "$file2" >$TMP_FILE2
        if diff -i -w $TMP_FILE1 $TMP_FILE2 >/dev/null
        then
            echo "$file1 is a copy of $file2"
        fi
    done
done
rm -f $TMP_FILE1 $TMP_FILE2
```

source code for plagiarism_detection.identifiers.sh

plagiarism detection - ignoring changes in code order

```
for file1 in "$@"
do
    for file2 in "$@"
    do
        test "$file1" = "$file2" &&
            break # avoid comparing pairs of assignments twice
        sed "$substitutions" "$file1"|sort >$TMP_FILE1
        sed "$substitutions" "$file2"|sort >$TMP_FILE2
        if diff -i -w $TMP_FILE1 $TMP_FILE2 >/dev/null
        then
            echo "$file1 is a copy of $file2"
        fi
    done
done
rm -f $TMP_FILE1 $TMP_FILE2
```

source code for plagiarism_detection.reordering.sh

robust creation & removal of temporary files

- our code can be more robust and more secure by using mktemp to generate temporary file names
- we can also use the builtin shell '**trap** command to ensure temporaryt files are removed however the script exits

```
TMP_FILE1=$(mktemp /tmp/plagiarism_tmp.1.XXXXXXXXXX)
TMP_FILE2=$(mktemp /tmp/plagiarism_tmp.2.XXXXXXXXXX)
trap 'rm -f $TMP_FILE1 $TMP_FILE2' INT TERM EXIT
```

source code for plagiarism_detection.mktemp.sh

- temporary file creation is major source of security holes by very careful creating temporary files
- find & use existing robust & well-tested code

plagiarism detection - using hashing

```
# Improved version of plagiarism_detection.reordering.sh
# Note use md5sum to calculate a Cryptographic hash of the modifiec
# http://en.wikipedia.org/wiki/MD5
# and  use of sort && uniq to find files with the same hash
# This allows execution time linear in the number of files
substitutions='
    s/\/\/.*//
    s/"[^"]"/s/g
    s/[a-zA-Z_][a-zA-Z0-9_]*/v/g'
for file in "$@"
do
    md5hash=$(sed "$substitutions" "$file"|sort|md5sum)
    echo "$md5hash $file"
done|
sort|
uniq -w32 -d --all-repeated=separate|
cut -c36-
```

source code for plagiarism_detection.md5_hash.sh

case statements in Shell

- shell case statements have this form:

case *word* **in**
pattern$_1$)
    commands$_1$
    ;;
pattern$_2$)
    commands$_2$
    ;;
pattern$_n$)
    commands~n
esac

- ***word*** is compared to each ***pattern***$_i$ in turn.
- for the first ***pattern***$_i$ that matches the corresponding ***commands***$_i$ is executed and the case statement finishes.

case statements in Shell

- case patterns use the same language as filename expansion (globbing)
  - in other words the special characters are `* ? []`
  - patterns are not interpreted as regexes
- shell programmer used to use **case** statements heavily for efficiency
  - much less important now and many shell programmers don't use case
  - but use of case can still make shell code more readable

case statement - examples

```
# Checking number of command line args
case $# in
0)  echo "You forgot to supply the argument" ;;
1)  filename=$1 ;;
*)  echo "You supplied too many arguments" ;;
esac

# Classifying a file via its name
case "$file" in
*.c)  echo "$file looks like a C source-code file" ;;
*.h)  echo "$file looks like a C header file" ;;
*.o)  echo "$file looks like a an object file" ;;
...
?)    echo "$file's name is too short to classify" ;;
*)    echo "I have no idea what $file is" ;;
esac
```


example - shell function

```
#!/bin/dash
# written by andrewt@unsw.edu.au for COMP(2041|9044)
# demonstrate simple use ofa shell function
repeat_message() {
    n=$1
    message=$2
    for i in $(seq 1 $n)
    do
        echo "$i: $message"
    done
}
i=0
while test $i -lt 4
do
    repeat_message 3 "hello Andrew"
    i=$((i + 1))
done
```

source code for repeat_message.sh

example - local variables in a shell function

```
# print print numbers < 10000
# note use of local Shell builtin to scope a variable
# without the local declaration
# the variable i in the function would be global
# and would break the bottom while loop
# local is not (yet) POSIX but is widely supported
is_prime() {
    local n i
    n=$1
    i=2
    while test $i -lt $n
    do
        test $((n % i)) -eq 0 &&
            return 1
        i=$((i + 1))
    done
    return 0
}
i=0
while test $i -lt 1000
do
    is_prime $i && echo $i
    i=$((i + 1))
done
```

source code for local.sh

example - catching a signal with trap

```
# catch signal SIGTERM, print message and exit
trap 'echo loop executed $n times in 1 second; exit 0' TERM
# launch a sub-shell that will terminate
# this process in 1 second
my_process_id=$$
(sleep 1; kill $my_process_id) &
n=0
while true
do
    n=$((n + 1))
done
```

source code for trap.sh

- EXPLANATION OF trap TO BE ADDED HERE

example - compiling in parallel

```
# compile  the .c files in the current directory in parallel
# assumes the .c files in the current directory are a single progra
# see create_1001_file_C_program.sh to create suitable test data
# On a CPU with n cores this may be close to n times faster
# But note if there are large number of C file we
# may exhaust memory or operating system resources
for f in *.c
do
    clang -c "$f" &
done
# wait for the incrementa compiles to finish
# and then produce binary
wait
clang *.o -o binary
```

source code for parallel_compile.v0.sh

example - compiling in parallel

```
# compile  the .c files in the current directory in parallel using
# assumes the .c files in the current directory are a single progra
# see create_1001_file_C_program.sh to create suitable test data
# on Linux getconf will tell us how many cores the machine has
# otherwise assume 8
max_processes=$(getconf _NPROCESSORS_ONLN 2>dev/null) ||
    max_processes=8
# NOTE: this breaks if a filename contains whitespace or quotes
ls *.c|
xargs --max-procs=$max_processes --max-args=1 clang -c
clang *.o -o binary
```

source code for parallel_compile.v1.sh

example - compiling in parallel

```
# compile  the .c files in the current directory in parallel using
# assumes the .c files in the current directory are a single progra
# see create_1001_file_C_program.sh to create suitable test data
# find's -print0 option terminates pathnames with a '\0'
# xargs's --null option expects '\0' terminated input
# as '\0' can not appear in file names this can handle any filename
# on Linux getconf will tell us how many cores the machine has
# otherwise assume 8
max_processes=$(getconf _NPROCESSORS_ONLN 2>dev/null) ||
    max_processes=8
find . -maxdepth 1 -type f -name '*.c' -print0|
xargs --max-procs=$max_processes --max-args=1  --null clang -c
clang *.o -o binary
```

source code for parallel_compile.v2.sh

example - compiling in parallel

```
# compile  the .c files in the current directory in parallel using
# assumes the .c files in the current directory are a single progra
# see create_1001_file_C_program.sh to create suitable test data
# find's -print0 option terminates pathnames with a '\0'
# parallel's --null option expects '\0' terminated input
# as '\0' can not appear in file names this can handle any filename
find . -maxdepth 1 -type f -name '*.c' -print0|
parallel --null clang -c '{}'
clang *.o -o binary
```

source code for parallel_compile.v3.sh

TBA

MORE TO BE ADDED HERE OR ABOVE INCLUDING:

- read command
- ${} syntax
- bash arithmetic extensions
- use of `set -x` for debugging
- an inotify example