Duration: **50 minutes**
Aids Allowed: **NONE**

Student Number: _____

Last (Family) Name(s): **SOLUTION**

First (Given) Name(s): _____

Tutorial Section:
(cirle one)

LM 157 James Li

SS 2128 Yuan Gao

---

*Do **not** turn this page until you have received the signal to start.*
(In the meantime, please fill out the identification section above, and read the instructions below *carefully*.)

This test consists of 4 questions on 7 pages (including this one), printed on one side of the paper. *When you receive the signal to start, please make sure that your copy of the test is complete.*

Answer each question directly on the test paper, in the space provided. If you need more space for one of your solutions, use the reverse side of a page and *indicate clearly the part of your work that should be marked.*

IMPORTANT: You do not need to include the "`#!`" line in Bourne shell scripts you are asked to write. In C programs, you do not need to add the "`#include`" lines, nor do error checking unless the question requires it or the program would not function correctly given valid input without error checking.

MARKING GUIDE

1: _____/ 8

2: _____/ 8

3: _____/ 5

4: _____/ 7

TOTAL: _____/28

Good Luck!

Question 1. [8 MARKS]

Consider the following Bourne shell script named `echo`.

```
#!/bin/sh
PATH=/bin:/usr/bin
echo "$1... $1... $1..."
```

Part (a) [1 MARK]

Alice puts the above `echo` script in the current working directory and runs the following command (`$` is the shell prompt):

```
$ echo hello
hello... hello... hello...
```

Explain why this output could be produced.

The echo script in the current working directory is executed, since . is in the PATH.

Part (b) [1 MARK]

Bob also puts the above `echo` script in the current working directory. He tries the same command, but gets the following output:

```
$ echo hello
hello
```

Explain why this output could be produced.

Either echo is a shell built-in, . is not in the PATH, or . appears after /bin (the location of the normal echo program) in the PATH.

Part (c) [1 MARK]

Does the above `echo` script call itself (is it recursive)? Explain why or why not.

No. The PATH is reset for the script, so the normal echo command will be executed within the script (assuming this script isn't put in `/bin` in place of the `echo` program).

Question 1. (CONTINUED)

Part (d) [5 MARKS]

Consider the following C declarations. Write the type of the expressions that follow, or write "invalid" if the expression is not legal in C. If the type is valid and not a pointer, then give the value of the expression.

```
struct tut {
  char b[10];
  int r;
};
struct tut a[3] = {{"LM", 157}, {"SS", 2128}};
struct tut *p = &a[1];
```

	Type	**Value**
`a`	struct tut*	—
`a[1].r`	int	2128
`a->b[0]`	char	'L'
`*p->b`	char	'S'
`p[-1].r`	int	157

Question 2. [8 MARKS]

Part (a) [3 MARKS]

I need help writing the autotesting scripts for assignment 2. I want to check if the output from a student's phonem program is the same as the output from my solution. But the words can be output in any order, so I can't simply compare the two outputs.

Help me by writing a Bourne shell script (we'll call it `samelines`) that takes two files as command line arguments, produces no output, and returns true if the two files contain the same lines (regardless of order), and returns false otherwise.

The command `sort filename` sorts the lines of `filename` and outputs the result on the standard output. The command `diff file1 file2` outputs the differences between the two files and returns an exit status of 0 if no differences were found, 1 if some differences were found, and 2 means trouble. You may use temporary files, and you may assume that your script is run correctly (that two file names are given and both files exist).

SAMPLE SOLUTION:

```
#!/bin/sh
sort $1 > tmp1
sort $2 > tmp2
diff tmp1 tmp2 > /dev/null
```

This doesn't clean up the temporary files. To clean up the temportary files, add the following lines.

```
status=$?
rm -f tmp1 tmp2
exit $status
```

Question 2. (CONTINUED)

Part (b) [5 MARKS]

Write a Bourne shell script that uses the `samelines` script from part (a) to compute how many tests a student's program passed. Your script will take a single argument: an integer N, the number of tests. You script will, for each i, $1 \leq i \leq N$, use `samelines` to compare the files `student.`i and `expected.`i. Your script will print a single number, the number of pairs of files that matched.

For example, if your script is executed with the argument "2", you will compare `student.1` with `expected.1` and compare `student.2` with `expected.2`, and print either 0, 1 or 2.

SAMPLE SOLUTION:

```
#!/bin/sh
total=0
i=1
N=$1
while [ $i -le $N ]; do
   ./samelines student.$i expected.$i
   if [ $? -eq 0 ]; then
      total=‘expr $total + 1‘
   fi
   i=‘expr $i + 1‘
done
echo $total
```

Question 3. [5 MARKS]

Rewrite the contents of the following C function using pointers and without using array notation and without using the variables `i` or `j`. You may not use any library function calls. You are permitted to change the pointers `dest` and `src`.

```
void mystrcat(char *dest, char *src)
{
   int i, j;
   for (i = 0; dest[i] != '\0'; i++)
      ; /* do nothing */
   for (j = 0; src[j] != '\0'; i++, j++)
      dest[i] = src[j];
   dest[i] = src[j];
}
```

SAMPLE SOLUTION:

```
void mystrcat(char *dest, const char *src)
{
  while (*dest)
    dest++;
  while (*src)
    *dest++ = *src++;
  *dest = *src;
}
```

Question 4. [7 MARKS]

Suppose the main contents of your `phonem.c` program are put into the following function:

`int print_phonem_matches(FILE *dict, const char *string)`.

The `print_phonem_matches()` function takes a pointer to the open dictionary file and a pointer to the string to be matched, outputs to stdout all words in the dictionary that match `string`, and returns the number of words that were matched.

We wish to compute all two-word combinations that match the input string. Insert below the C code to complete this task. Assume that the correct dictionary has been opened, the string is valid, and `MAXLENGTH` is sufficiently large.

```
const char *string;  /* preset to contents of string */
FILE *dict;          /* dictionary is already fopened */
char substr1[MAXLENGTH];  /* copy first piece of string here */
char substr2[MAXLENGTH];  /* copy rest of string here */

  /* sample solution */
  int i;
  int len = strlen(string);
  for (i = 1; i < len-1; i++) {
       strncpy(substr1, string, i);  /* copy prefix of string */
       substr1[i] = '\0';            /* and null terminate */
       strcpy(substr2, string+i);  /* copy rest of string into second substring */

         printf("\nPossible matches for %s-%s\n", substr1, substr2);
         printf("---Possible first words:\n");
         rewind(dict);   /* rewind starts reading from start of dict */
         if (print_phonem_matches(dict, substr1) == 0)
            printf("no words matched %s\n", substr1);
         printf("---Possible second words:\n");
         rewind(dict);   /* reset to start of dictionary again */
         if (print_phonem_matches(dict, substr2) == 0)
            printf("no words matched %s\n", substr2);

  }
```

C functions for strings:

```
size_t strlen(const char *s);
int strncmp(const char *s1, const char *s2, size_t n);
char *strncpy(char *dest, const char *src, size_t n);
char *strncat(char *dest, const char *src, size_t n);
char *index(const char *s, int c);
char *strchr(const char *s, int c);
char *strstr(const char *haystack, const char *needle);
```

C functions for files and directories:

```
int closedir(DIR *dir);
int fclose(FILE *stream);
char *fgets(char *s, int n, FILE *stream);
FILE *fopen(const char *file, const char *mode);
int fprintf(FILE *stream, const char *format, ...);
char *getcwd(char *buf, size t size);
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
int stat(const char *file name, struct stat *buf);
void perror(const char *s);
```

```
struct stat {
  dev_t         st_dev;     /* device */
  ino_t         st_ino;     /* inode */
  mode_t        st_mode;    /* protection */
  nlink_t       st_nlink;   /* number of hard links */
  off_t         st_size;    /* total size, in bytes */
  unsigned long st_blksize; /* blocksize for filesystem I/O */
  unsigned long st_blocks;  /* number of blocks allocated */
  time_t        st_atime;   /* time of last access */
  time_t        st_mtime;   /* time of last modification */
  time_t        st_ctime;   /* time of last change */
};
```

The following POSIX macro functions are defined to check the file type (`m` is the `st_mode` field of the stat struct):
`S_ISLNK(m)` is it a symbolic link?
`S_ISREG(m)` regular file?
`S_ISDIR(m)` directory?

Shell variables:
`$$` shell process ID
`$?` last program exit status
`$#` number of arguments
`$*` all arguments as string
`"$@"` all arguments as quoted list

Shell test comparison operators:

Shell	Description
-d filename	Exists as a directory
-f filename	Exists as a regular file
-r filename	Exists as a readable file
-w filename	Exists as a writable file
-x filename	Exists as an executable file
-z string	True if empty string
str1 = str2	True if str1 equals str2
str1 != str2	True if str1 not equal to str2
int1 -eq int2	True if int1 equals int2
-ne, -gt, -ge, -lt, -le	Comparisons for numbers
!=, >, >=, <, <=	Comparisons for strings
-a, -o	And, or