20T2 Final Exam Sample Answers

Exam Conditions

- You can start reading this exam at **Saturday 28 November 12:50** Sydney time.
- You can start typing at **Saturday 28 November 13:00** Sydney time.
- You have until **Saturday 28 November 16:00** Sydney time to complete this exam
- Only submissions before **Saturday 28 November 16:00** Sydney time will be marked
- Except, students with extra exam time approved by Equitable Learning Services (ELS) can make submissions after **Saturday 28 November 16:00** within their approved extra time
- You are not permitted to communicate (email, phone, message, talk, ...) with anyone during this exam, except COMP1521 staff via **cs1521.exam@cse.unsw.edu.au**
- You are not permitted to get help from anyone but COMP1521 staff during this exam.
- This is a closed book exam.
- You are not permitted to access papers or books.
- You are not permitted to access files on your computer or other computers, except the files for the exam.
- You are not permitted to access web pages or other internet resources, except the web pages for the exam and the online language cheatsheets & documentation linked below
- **Deliberate violation of exam conditions will be referred to Student Integrity as serious misconduct**

Exam Structure

- There are **11** questions on this exam.
- Total mark of questions on this exam is **100**.
- Questions are **NOT** worth equal marks.
- All 11 questions are practical (programming) questions.
- Not all questions may have provided files, You should create any files needed for submission if they are not provided.
- Answer each question in a **SEPARATE** file. Each question specifies the name of the file to use. These are named after the corresponding question number, Make sure you use **EXACTLY** this file name.
- When you finish working on a question, submit the files using the **give** command provided in the question. **You may submit your answers as many times as you like.** The last submission **ONLY** will be marked.
- Do not leave it to the deadline to submit your answers. Submit each question when you finish working on it. Running autotests does not automatically submit your code.
- You can verify what submissions you have made with `1521 classrun -check practice_q<N>`

Language Documentation

You may access this **language documentation** while attempting this test:

- C quick reference
- MIPS Quick Reference Card
- MIPS Instruction Reference
- SPIM Documentation
- MIPS Quick Tutorial

You may also access:

- manual entries via the `man` command
- Texinfo pages via the `info` command

Special Considerations

> This exam is covered by the Fit-to-Sit policy. That means that by sitting this exam, you are declaring yourself well enough to do so. You will be unable to apply for special consideration after the exam for circumstances affecting you before it began. If you have questions, or you feel unable to complete the exam, contact **cs1521.exam@cse.unsw.edu.au**

If you experience a technical issue before or during the exam, you should follow the following instructions:

Take screenshots of as many of the following as possible:

- error messages

- error messages
- screen not loading
- timestamped speed tests
- power outage maps
- messages or information from your internet provider regarding the issues experienced

You should then get in touch with course staff via **cs1521.exam@cse.unsw.edu.au** as soon as the issue arises

Getting Started

Set up for the exam by creating a new directory called `exam_practice`, changing to this directory, and fetching the provided code by running these commands:

```
$ mkdir -m 700 exam_practice
$ cd exam_practice
$ 1521 fetch exam_practice
```

Or you can download the provided code as a zip file or a tar file.

If you make a mistake and need a new copy of a particular file you can do the follow:

```
$ rm broken-file
$ 1521 fetch exam_practice
```

Only files that don't exist will be recreated, all other files will remain untouched

Question 1 (10 MARKS)

You have been given `practice_q1.s`, a MIPS assembler program that reads one number and then prints it.

Add code to `practice_q1.s` to make it equivalent to this C program:

```c
// print the sum of two integers

#include <stdio.h>

int main(void) {
    int x, y;

    scanf("%d", &x);
    scanf("%d", &y);
    printf("%d\n", x + y);

    return 0;
}
```

In other words, it should read 2 numbers and print their sum.

For example:

```
$ 1521 spim -f practice_q1.s
5
8
13
$ 1521 spim -f practice_q1.s
118
26
144
$ 1521 spim -f practice_q1.s
42
42
84
```

NOTE:

No error checking is required.

Your program can assume its input always contains two integers, and only two integers.

You can assume the value of the expression can be represented as a signed 32 bit value. In other words, you can assume overflow/underflow does not occur.

Your solution must be in MIPS assembler only.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest practice_q1
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 practice_q1 practice_q1.s
```

To verify your submissions for this activity:

```
$ 1521 classrun -check practice_q1
```

Sample solution for `practice_q1.s`

```
#  print the sum of two integers
# x in $t0, y in $t1
main:
    li $v0, 5           #   scanf("%d", &x);
    syscall             #
    move $t0, $v0

    li $v0, 5           #   scanf("%d", &y);
    syscall             #
    move $t1, $v0


    add $a0, $t0, $t1   #   z = x + y
    li $v0, 1           #   printf("%d", z);
    syscall

    li   $a0, '\n'      #   printf("%c", '\n');
    li   $v0, 11
    syscall

end:

    li $v0, 0           # return 0
    jr $31
```

Question 2 (9 MARKS)

Your task is to add code to this function in **practice_q2.c**:

```
// given a uint32_t value,
// return 1 iff the least significant (bottom) byte
// is equal to the 2nd least significant byte; and
// return 0 otherwise
int practice_q2(uint32_t value) {
    // PUT YOUR CODE HERE

    return 42;
}
```

Add code to the function `practice_q2` so that, given a `uint32_t` value, it returns **1** iff (if and only if) the least significant (bottom) byte of `value` is equal to the second least significant byte. `practice_q2` should return **0** otherwise.

For example, given the hexadecimal value **0x12345678**, `practice_q2` should return **0**, because the least significant byte, **0x78**, *is not* equal to the second least significant byte, **0x56**.

Similarly, given the hexadecimal value **0x12345656**, `practice_q2` should return **1**, because the least significant byte, **0x56**, *is* equal to the second least significant byte, **0x56**.

You must use bitwise operators to implement `practice_q2`.

For example:

```
$ dcc practice_q2.c test_practice_q2.c -o practice_q2
$ ./practice_q2 0x00000000
practice_q2(0x00000000) returned 1
$ ./practice_q2 0x00000001
practice_q2(0x00000001) returned 0
$ ./practice_q2 0x00000100
practice_q2(0x00000100) returned 0
$ ./practice_q2 0x00000101
practice_q2(0x00000101) returned 1
$ ./practice_q2 0x00001212
practice_q2(0x00001212) returned 1
$ ./practice_q2 0x00001213
practice_q2(0x00001213) returned 0
$ ./practice_q2 0x12345678
practice_q2(0x12345678) returned 0
$ ./practice_q2 0x12345656
practice_q2(0x12345656) returned 1
$ ./practice_q2 0x12345634
practice_q2(0x12345634) returned 0
$ ./practice_q2 0x12345666
practice_q2(0x12345666) returned 0
$ ./practice_q2 0x12121212
practice_q2(0x12121212) returned 1
$ ./practice_q2 0x37861212
practice_q2(0x37861212) returned 1
$ ./practice_q2 0x12121221
practice_q2(0x12121221) returned 0
```

You can also use *make* to build your code:

```
$ make practice_q2
```

NOTE:

You are not permitted to call any functions from the C standard library.

You are not permitted to use division (/), multiplication (*), or modulus (%).

You are not permitted to change the `main` function you have been given.

You are not permitted to change `practice_q2`'s prototype (its return type and argument types).

No error checking is necessary.

You may define and call your own functions if you wish.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest practice_q2
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 practice_q2 practice_q2.c
```

To verify your submissions for this activity:

```
$ 1521 classrun -check practice_q2
```

Sample solution for `practice_q2.c`

```c
// Sample solution

#include <stdint.h>
#include <stdlib.h>
#include <assert.h>

// given uint32_t value return 1 iff
// the least significant (bottom) byte
// is equal to the 2nd least significant byte
// return 0 otherwise
int practice_q2(uint32_t value) {
    uint32_t bottom_byte = value & 0xFF;
    uint32_t second_byte = (value >> 8) & 0xFF ;
    return bottom_byte == second_byte;
}
```

Question 3 (9 MARKS)

You have been given `practice_q3.s`, a MIPS assembler program that reads an integer value and then prints it.

Add code to the file `practice_q3.s` so that it prints **1** iff the least significant (bottom) byte of value is equal to the second least significant byte, and prints **0** otherwise.

For example, given the decimal value **305419896**, which is hexadecimal **0x12345678**, `practice_q3.s` should print **0**, because the least significant byte, **0x78**, *is not* equal to the second least significant byte, **0x56**.

Similarly, given the decimal value **305419862**, which is hexadecimal **0x12345656**, `practice_q3.s` should print **1**, because the least significant byte, **0x56**, *is* equal to the second least significant byte, **0x56**.

For example:

```
$ 1521 spim -f practice_q3.s
0
1
$ 1521 spim -f practice_q3.s
1
0
$ 1521 spim -f practice_q3.s
256
0
$ 1521 spim -f practice_q3.s
257
1
$ 1521 spim -f practice_q3.s
4626
1
$ 1521 spim -f practice_q3.s
4627
0
$ 1521 spim -f practice_q3.s
305419896
0
$ 1521 spim -f practice_q3.s
305419862
1
```

NOTE:

Your solution must be in MIPS assembler only.

Your program can assume its input always contains one integer.

No error checking is necessary.

It is recommended, but not required, that you use bitwise operators.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest practice_q3
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 practice_q3 practice_q3.s
```

To verify your submissions for this activity:

```
$ 1521 classrun -check practice_q3
```

Sample solution for `practice_q3.s`

```
main:
    li   $v0, 5
    syscall

    andi $t0, $v0, 0xff
    srl  $t1, $v0, 8
    andi $t1, $t1, 0xff

    seq  $a0, $t0, $t1
    move $a0, $a0
    li   $v0, 1
    syscall

    li   $a0, '\n'
    li   $v0, 11
    syscall

end:
    li   $v0, 0
    jr   $31
```

Question 4 (9 MARKS)

You have been given `practice_q4.s`, a MIPS assembler program that reads 1 number and then prints it.

Add code to `practice_q4.s` to make it equivalent to this C program:

```c
//  read numbers until their sum is >= 42, print their sum

#include <stdio.h>

int main(void) {
    int sum = 0;
    while (sum < 42) {
        int x;
        scanf("%d", &x);
        sum = sum + x;
    }
    printf("%d\n", sum);
    return 0;
}
```

In other words, it should read numbers until their sum is ≥>= 42 and then print their sum.

For example:

```
$ 1521 spim -f practice_q4.s
10
20
25
55
$ 1521 spim -f practice_q4.s
20
22
42
$ 1521 spim -f practice_q4.s
100
100
$ 1521 spim -f practice_q4.s
10
10
10
10
10
50
```

NOTE:

No error checking is required.

Your program can assume its input contains only integers.

Your program can assume these integers sum to >at least 42

Your program can assume these integers sum to ≥at least 42.

You can assume the value of the expression can be represented as a signed 32 bit value. In other words, you can assume overflow/underflow does not occur.

Your solution must be in MIPS assembler only.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest practice_q4
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 practice_q4 practice_q4.s
```

To verify your submissions for this activity:

```
$ 1521 classrun -check practice_q4
```

Sample solution for `practice_q4.s`

```
# read integers until their sum is >= 42, print their sum
# sum in $t0
main:
    li $t0, 0
loop:
    li $v0, 5            # scanf("%d", &x);
    syscall              #
    add $t0, $t0, $v0
    blt $t0, 42, loop

    move $a0, $t0        # printf("%d", x);
    li $v0, 1
    syscall

    li   $a0, '\n'       # printf("%c", '\n');
    li   $v0, 11
    syscall

    li $v0, 0            # return 0
    jr $31
```

Question 5 (9 MARKS)

Write a C program, `practice_q5.c`, which takes two names of environment variables as arguments. It should print **1** iff both environment variables are set to the same value. Otherwise, if the environment variables differ in value, or either environment variable is not set, it should print **0**.

The shell command `export` sets an environment variable to a value; the shell command `unset` unsets an environment variable. In the following example, `export` is used to set the environment variables `VAR1`, `VAR2` and `VAR3`, and `unset` is used to ensure environment variables `VAR4` and `VAR5` are unset.

```
$ export VAR1=hello
$ export VAR2=good-bye
$ export VAR3=hello
$ unset VAR4
$ unset VAR5
$ ./practice_q5 VAR1 VAR2
0
$ ./practice_q5 VAR3 VAR1
1
$ ./practice_q5 VAR2 VAR4
0
$ ./practice_q5 VAR4 VAR5
0
```

NOTE:

There is no supplied code for this question.

Your program can assume it is always given 2 arguments.

Your program should always print one line of output. The line of output should contain only 0 or 1.

Your solution must be in C only.

You are not permitted to run external programs. You are not permitted to use system, popen, posix_spawn, fork or exec.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest practice_q5
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 practice_q5 practice_q5.c
```

To verify your submissions for this activity:

```
$ 1521 classrun -check practice_q5
```

Sample solution for `practice_q5.c`

```c
// test the value of two environment are the same
// if so print 1, else print 0

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

int main(int argc, char *argv[]) {
    assert(argc == 3);
    char *value1 = getenv(argv[1]);
    char *value2 = getenv(argv[2]);
    if (value1 != NULL && value2 != NULL && strcmp(value1, value2) == 0) {
        printf("1\n");
    } else {
        printf("0\n");
    }
    return 0;
}
```

Question 6 (9 MARKS)

We need to count the number of ASCII bytes in files.

Write a C program, `practice_q6.c`, which takes a single filename as its argument, counts the number of bytes in the named file which are valid ASCII, and prints one line of output containing that count.

Assume a byte is valid ASCII iff it contains a value between 0 and 127 inclusive.

- You must match the output format in the example below exactly.

```
$ dcc practice_q6.c -o practice_q6
$ echo hello world >file1
$ ./practice_q6 file1
file1 contains 12 ASCII bytes
$  echo -e 'hello\xBAworld' >file2
$ ./practice_q6 file2
file2 contains 11 ASCII bytes
$ echo -n -e '\x80\x81' >file3
$ ./practice_q6 file3
file3 contains 0 ASCII bytes
```

NOTE:

There is no supplied code for this question.

No error checking is required.

Your program can assume it is always given the name of a file.

Your solution must be in C only.

You are not permitted to run external programs. You are not permitted to use system, popen, posix_spawn, fork or exec.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest practice_q6
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 practice_q6 practice_q6.c
```

To verify your submissions for this activity:

```
$ 1521 classrun -check practice_q6
```

Sample solution for `practice_q6.c`

```c
// Print position of first non-ASCII byte in file

#include <stdio.h>
#include <stdlib.h>

void process_file(char *pathname);

int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        process_file(argv[arg]);
    }
    return 0;
}

void process_file(char *pathname) {
    FILE *stream = fopen(pathname, "r");
    if (stream == NULL) {
        perror(pathname);
        exit(1);
    }

    ssize_t ascii_count = 0;
    int byte;
    while ((byte = fgetc(stream)) != EOF) {
        if (byte < 128) {
             ascii_count++;
        }
     }

    fclose(stream);
    printf("%s contains %zd ASCII bytes\n",  pathname, ascii_count);
}
```

Question 7 (9 MARKS)

You have been given `practice_q7.s`, a MIPS assembler program that reads 1 number and then prints it.

Add code to `practice_q7.s` to make it equivalent to this C program:

```c
// Read numbers into an array until their sum is >= 42
// then print the numbers in reverse order

#include <stdio.h>

int numbers[1000];

int main(void) {
    int i = 0;
    int sum = 0;
    while (sum < 42) {
        int x;
        scanf("%d", &x);
        numbers[i] = x;
        i++;
        sum += x;
    }

    while (i > 0) {
        i--;
        printf("%d\n", numbers[i]);
    }
}
```

In other words, it should read numbers until their sum is ≥ 42, and then print the numbers read in reverse order.

For example:

```
$ 1521 spim -f practice_q7.s
11
17
3
19
19
3
17
11
$ 1521 spim -f practice_q7.s
10
20
30
30
20
10
$ 1521 spim -f practice_q7.s
42
42
```

No error checking is required.

Your program can assume its input contains only integers.

Your program can assume these integers sum to ≥ 42.

Your program can assume that it will have to read no more than 1000 integers before their sum is ≥ 42.

You can assume the value of the expression can be represented as a signed 32 bit value. In other words, you can assume overflow/underflow does not occur.

Your solution must be in MIPS assembler only.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest practice_q7
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 practice_q7 practice_q7.s
```

To verify your submissions for this activity:

```
$ 1521 classrun -check practice_q7
```

Sample solution for `practice_q7.s`

```
# Read numbers into an array until their sum is >= 42
# then print the numbers in reverse order

# i in register $t0
# registers $t1, $t2 & $t3 used to hold temporary results

main:
    li $t0, 0           # i = 0
    li $t4, 0           # i = 0
loop0:
    bge $t4, 42, end0 # while (i < 1000) {

    li $v0, 5           #   scanf("%d", &numbers[i]);
    syscall             #

    blt $v0, 0, end0    # if (x < 0) break
    mul $t1, $t0, 4     #   calculate &numbers[i]
    la $t2, numbers     #
    add $t3, $t1, $t2   #
    sw $v0, ($t3)       #   store entered number in array
    add $t4, $t4, $v0
    add $t0, $t0, 1     #   i++;
    b loop0             # }
end0:

loop1:
    ble $t0, 0, end1   # while (i > 0) {

    add $t0, $t0, -1    #   i--

    mul $t1, $t0, 4     #   calculate &numbers[i]
    la $t2, numbers     #
    add $t3, $t1, $t2   #
    lw $a0, ($t3)       #   load numbers[i] into $a0

    li $v0, 1           #   printf("%d", numbers[i])
    syscall

    li   $a0, '\n'      #   printf("%c", '\n');
    li   $v0, 11
    syscall

    b loop1             # }
end1:

    li $v0, 0           # return 0
    jr $31

.data
numbers:
    .space 4000
```

Question 8 (9 MARKS)

We need to count the number of UTF-8 characters in a file, and check that the file contains only valid UTF-8.

Write a C program, `practice_q8.c`, which takes a single filename as its argument, counts the number of UTF-8 characters in the named file, and prints one line of output containing that count.

Use the same format as the example below.

```
$ dcc practice_q8.c -o practice_q8
$ echo hello world >file1
$ ./practice_q8 file1
file1: 12 UTF-8 characters
$ echo -e -n '\xF0\x90\x8D\x88' >file2
$ ./practice_q8 file2
file2: 1 UTF-8 characters
$ echo -e -n '\x24\xC2\xA2\xE0\xA4\xB9' >file3
$ ./practice_q8 file3
file3: 3 UTF-8 characters
$ ./practice_q8 utf8.html
utf8.html: 7943 UTF-8 characters
```

If `practice_q8.c` reads a byte which is not valid UTF-8, it should stop and print an error message in the same format as the example below. Note the error message includes how many valid UTF-8 characters have been previously read.

```
$ echo -e -n '\x24\xC2\xA2\xE0\xA4\x09' >file4
$ ./practice_q8 file4
file4: invalid UTF-8 after 2 valid UTF-8 characters
```

If the end of file is reached before the completion of a UTF-8 character, `practice_q8.c` should also print an error message; for example:

```
$ echo -e -n '\x24\xC2\xA2\xE0\xA4' >file5
$ ./practice_q8 file5
file5: invalid UTF-8 after 2 valid UTF-8 characters
```

A reminder of how UTF-8 is encoded:

#bytes	*#bits*	*Byte 1*	*Byte 2*	*Byte 3*	*Byte 4*
1	7	`0xxxxxxx`	-	-	-
2	11	`110xxxxx`	`10xxxxxx`	-	-
3	16	`1110xxxx`	`10xxxxxx`	`10xxxxxx`	-
4	21	`11110xxx`	`10xxxxxx`	`10xxxxxx`	`10xxxxxx`

NOTE:

There is no supplied code for this question.

Only the specified error checking is required; no additional error checking is required.

Your program can assume it is always given a single argument, the name of a file.

Your solution must be in C only.

You are not permitted to run external programs. You are not permitted to use system, popen, posix_spawn, fork or exec.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest practice_q8
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 practice_q8 practice_q8.c
```

To verify your submissions for this activity:

```
$ 1521 classrun -check practice_q8
```

Sample solution for `practice_q8.c`

```c
// Print position of first non-ASCII byte in file

#include <stdio.h>
#include <stdlib.h>

void process_file(char *pathname);

int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        process_file(argv[arg]);
    }
    return 0;
}

void invalid(char *pathname, ssize_t utf8_count) {
    printf("%s: invalid UTF-8 after %zd valid UTF-8 characters\n",  pathname, utf8_count);
    exit(0);
}

int get_continuation_byte(FILE *stream, char *pathname, ssize_t utf8_count) {
    int byte = fgetc(stream);
    if (byte == EOF || (byte & 0xC0) != 0x80) {
        invalid(pathname,  utf8_count);
    }
    return byte;
}

void process_file(char *pathname) {
    FILE *stream = fopen(pathname, "r");
    if (stream == NULL) {
        perror(pathname);
        exit(1);
    }

    ssize_t utf8_count;
    int byte1;
    for (utf8_count = 0; (byte1 = fgetc(stream)) != EOF; utf8_count++) {
        if ((byte1 & 0x80) == 0x00) {
            continue;
        }

        get_continuation_byte(stream, pathname, utf8_count);

        if ((byte1 & 0xE0) == 0xC0) {
            continue;
        }

        get_continuation_byte(stream, pathname, utf8_count);

        if ((byte1 & 0xF0) == 0xE0) {
            continue;
        }

        get_continuation_byte(stream, pathname, utf8_count);

        if ((byte1 & 0xF8) == 0xF0) {
            continue;
        }

        invalid(pathname,  utf8_count);
    }

    fclose(stream);
    printf("%s: %zd UTF-8 characters\n",  pathname, utf8_count);
}
```

Question 9 (9 MARKS)

You have been given `practice_q9.s`, a MIPS assembler program that reads a line of input and then prints 42.

Add code to `practice_q9.s` to make it equivalent to this C program:

```c
#include <stdio.h>

char *s;

int expression(void);
int term(void);
int number(void);

int main(int argc, char *argv[]) {
    char line[10000];
    fgets(line, 10000, stdin);
    s = line;
    printf("%d\n", expression());
    return 0;
}

int expression(void) {
    int left = term();
    if (*s != '+') {
        return left;
    }
    s++;
    int right = expression();
    return left + right;
}


int term(void) {
    int left = number();
    if (*s != '*') {
        return left;
    }
    s++;
    int right = term();
    return left * right;
}

int number(void) {
    int n = 0;
    while (*s >= '0' && *s <= '9') {
        n = 10 * n + *s - '0';
        s++;
    }
    return n;
}
```

The above C program reads a line of input containing an arithmetic expression and prints its value.

The arithmetic expression contains only positive integers, and multiply (`'*'`)and add (`'+'`) operators.

For example:

```
$ 1521 spim -f practice_q9.s
6*7
42
$ 1521 spim -f practice_q9.s
1+2*3*4+5
30
$ 1521 spim -f practice_q9.s
100+2*20+978
1118
$ 1521 spim -f practice_q9.s
1000+777+66+55*444+9
26272
```

NOTE:

No error checking is required.

Your program can assume it is given *exactly* one line of input.

Your program can assume this line contains only these 12 characters: **0123456789+***

You can assume the line contains less than 10,000 characters.

Your program can assume that these characters form a valid arithmetic expression.

You can assume the value of the expression can be represented as a signed 32 bit value. In other words you can assume overflow does not occur.

Your solution must be in MIPS assembler only.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest practice_q9
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 practice_q9 practice_q9.s
```

To verify your submissions for this activity:

```
$ 1521 classrun -check practice_q9
```

Sample solution for `practice_q9.s`

```
# this code reads a line of input and prints 42
# change it to evaluate the arithmetic expression

main:
      sw    $fp, -4($sp)
      la    $fp, -4($sp)
      sw    $ra, -4($fp)
      addi $sp, $sp, -8

    la   $a0, line
    la   $a1, 10000
    li   $v0, 8                         # fgets(line, 10000, stdin)
    syscall

    la $t0, line
    jal expression

    move $a0, $v0                       # printf("%d", expression());
    li   $v0, 1
    syscall

    li   $a0, '\n'                       # printf("%c", '\n');
    li   $v0, 11
    syscall

      lw $ra, -4($fp)
      la $sp, 4($fp)
      lw $fp, ($fp)

    li   $v0, 0                         # return 0;
    jr   $31

expression:
      sw    $fp, -4($sp)
    la  $fp, -4($sp)
    sw  $ra, -4($fp)
    sw  $s0, -8($fp)
    sw  $s1, -12($fp)
    addi $sp, $sp, -16

    jal term
    move $s1, $v0                       # left = term();

    lb $t4, ($t0)
    bne $t4, '+', expression_left       # if (*s != '+') {
    j expression_right                  #

expression_left:

    move $v0, $s1                       # return left;
    j expression_ep                     # }

expression_right:
    addi $t0, $t0, 1                    # s++;

    jal expression
    move $s2, $v0 #s2 = right           # right = expression();

    add $v0, $s1, $s2                   # return left + right;

expression_ep:
    lw $s1, -12($fp)
    lw $s0, -8($fp)
    lw $ra, -4($fp)
    la $sp, 4($fp)
    lw $fp, ($fp)

    jr $ra

term:
    sw  $fp, -4($sp)
    la  $fp, -4($sp)
    sw  $ra, -4($fp)
    sw  $s0, -8($fp)
    sw  $s1, -12($fp)
```

```
    sw  $s1, -12($fp)
    addi $sp, $sp, -16

    jal number
    move $s1, $v0 #s1 = left        # left = number();

    lb $t4, ($t0)
    bne $t4, '*', term_left         # if (*s != '*') {
    j term_right

term_left:
    move $v0, $s1                   # return left;
    j term_ep                       # }

term_right:
    addi $t0, $t0, 1                # s++;
    jal term
    move $s2, $v0 #s2 = right       # right = term();
    mul $v0, $s1, $s2               # return left * right;

term_ep:
    lw $s1, -12($fp)
    lw $s0, -8($fp)
    lw $ra, -4($fp)
    la $sp, 4($fp)
    lw $fp, ($fp)

    jr $ra

number:
    sw  $fp, -4($sp)
    la  $fp, -4($sp)
    sw  $ra, -4($fp)
    addi $sp, $sp, -8

    li $t3, 0                       # n = 0;

while:                              # while (*s >= '0' && *s <= '9') {
    lb $t4, ($t0)
    blt $t4, '0', end_while
    bgt $t4, '9', end_while

    mul $t3, $t3, 10                # n = n * 10;
    add $t3, $t3, $t4               # n = n + *s;
    sub $t3, $t3, '0'               # n = n - '0';
    addi $t0, $t0, 1                # s++;

    j while
end_while:                          # }

    lw $ra, -4($fp)
    la $sp, 4($fp)
    lw $fp, ($fp)

    move $v0, $t3                   # return n;

    jr $ra


.data
line:
    .space 10000
```

Question 10 (9 MARKS)

We need a program which will save an entire directory tree as a single file. The directory tree may consist of many directories and files.

Write a C program, `practice_q10`, which is given either 1 or 2 arguments.

If practice_q10 is given 2 arguments, the first argument will be the pathname of a file it should create and the second argument will be the pathname of a directory. practice_q10 should then save the entire contents of the specified directory tree in the specified file.

If practice_q10 is given 1 argument, that argument will be the pathname of a file in which a directory tree has been saved. practice_q10 should re-create all the directories and files in the directory tree.

For example, these commands create a directory tree named **a**:

```
$ mkdir -p a/b/c
$ echo hello andrew >a/file1
$ echo bye andrew >a/b/file2
$ echo 1 >a/b/c/one
$ echo 2 >a/b/c/two
$ ls -lR a
a:
total 8
drwxr-xr-x 3 z1234567 z1234567 4096 Aug 19 20:38 b
-rw-r--r-- 1 z1234567 z1234567   13 Aug 19 20:38 file1

a/b:
total 8
drwxr-xr-x 2 z1234567 z1234567 4096 Aug 19 20:38 c
-rw-r--r-- 1 z1234567 z1234567   11 Aug 19 20:38 file2

a/b/c:
total 8
-rw-r--r-- 1 z1234567 z1234567 2 Aug 19 20:38 one
-rw-r--r-- 1 z1234567 z1234567 2 Aug 19 20:38 two
$ cat a/file1
hello andrew
$ cat a/b/file2
bye andrew
```

In this example, practice_q10 saves the contents of the directory tree **a** into a file named **data**:

```
$ dcc practice_q10.c -o practice_q10
$ ./practice_q10 data a
$ ls -l data
-rw-r--r-- 1 z1234567 z1234567 4567 Aug 19 20:38 data
```

This example shows practice_q10 restoring the contents of the directory tree **a** after it has been removed:

```
$ rm -rf a
$ ls -lR a
ls: cannot access 'a': No such file or directory
$ cat a/file1
cat: a/file1: No such file or directory
$ ./practice_q10 data
$ ls -lR a
a:
total 8
drwxr-xr-x 3 z1234567 z1234567 4096 Aug 19 20:38 b
-rw-r--r-- 1 z1234567 z1234567   13 Aug 19 20:38 file1

a/b:
total 8
drwxr-xr-x 2 z1234567 z1234567 4096 Aug 19 20:38 c
-rw-r--r-- 1 z1234567 z1234567   11 Aug 19 20:38 file2

a/b/c:
total 8
-rw-r--r-- 1 z1234567 z1234567 2 Aug 19 20:38 one
-rw-r--r-- 1 z1234567 z1234567 2 Aug 19 20:38 two
$ cat a/file1
hello andrew
$ cat a/b/file2
bye andrew
```

This example shows practice_q10 restoring the contents of the directory tree **a** in a different directory:

```
$ mkdir new_directory
$ cd new_directory
$ ../practice_q10 ../data
$ cat a/file1
hello andrew
$ cat a/b/file2
bye andrew
```

WARNING

Autotest will only be of limited assistance in debugging your program. Do not expect autotest messages to be easy to understand for this problem. You will need to debug your program yourself.

DANGER:

It is easily possible to destroy many files with *rm*. Do not test this on your working files for this exam, unless you have a backup. Do not assume your program will make a good enough backup.

NOTE:

Your solution must be in C only.

You are not permitted to run external programs. You are not permitted to use system, popen, posix_spawn, fork or exec.

You are not permitted to use libraries other than the default C libraries. In other words your solution can not require use of dcc's -l flag. If your solution compiles without dcc's -l flag, you are using only the default C libraries. All functions discussed in lectures are part of the default C libraries.

No error checking is necessary.

You can assume the directory tree to be saved contains only directories and regular files. You can assume it does not contain links or other special files. You can assume it does not contain sparse files.

You can not assume anything about the size or contents of files in the directory tree. The files may contain any byte. The files may be any size.

Your program does not have to save or restore permissions, modification times, or other file metadata.

You can assume the directory tree to be saved contains at most 10000 directories and regular files.

You can assume the directory tree to be saved is at most 1000 levels deep.

You can assume files and directories do not already exist when restoring a directory tree.

You can use any format you choose to save the directory tree in the file.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest practice_q10
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 practice_q10 practice_q10.c
```

To verify your submissions for this activity:

```
$ 1521 classrun -check practice_q10
```

Sample solution for `practice_q10.c`

```c
// THIS IS NOT A VALID SOLUTION TO THIS QUESTION
// IT RUNS THE EXTERNAL PROGRAM TAR WHICH IS FORBIDDEN BY THE QUESTION

// This problem is a subset of 20T3 assignment 2
// what is required is either blobby -x or blobby -c
// but the blob format can be simpler

#include <stdio.h>
#include <stdlib.h>
#include <spawn.h>
#include <sys/wait.h>
#include <assert.h>

int main(int argc, char *argv[]) {
    char *tar_argv[5];
    tar_argv[0] = "/bin/tar";
    if (argc == 3) {
        tar_argv[1] = "cf";
        tar_argv[2] = argv[1];
        tar_argv[3] = argv[2];
        tar_argv[4] = NULL;
    } else {
        assert(argc == 2);
        tar_argv[1] = "xf";
        tar_argv[2] = argv[1];
        tar_argv[3] = NULL;
    }
    pid_t pid;
    extern char **environ;
    if (posix_spawn(&pid, "/bin/tar", NULL, NULL, tar_argv, environ) != 0) {
        perror("spawn");
        exit(1);
    }

    int exit_status;
    if (waitpid(pid, &exit_status, 0) == -1) {
        perror("waitpid");
        exit(1);
    }
    return exit_status;
}
```

Question 11 (9 MARKS)

You have been given `practice_q11.s` a MIPS assembler program that reads a line of input and then prints 42.

You have been given `practice_q11.s`, a MIPS assembler program that reads a line of input and then prints 42.

Add code to `practice_q11.s` to evaluate the line as an arithmetic expression. and print its value.

The arithmetic expression will contain only positive integers and multiply (`'*'`) and add (`'+'`) operators.

The integers may be arbitrarily large; except they must fit on a line of less than 10,000 characters.

There are no marks for approaches which handle only smaller integers; for example, those small enough to be represented in only 32 or 64 bits.

There are no marks for approaches which use floating point arithmetic or other methods to produce approximate results.

For example:

```
$ 1521 spim -f practice_q11.s
982451653*961748941*472882027*452390477
2021339052439370866631747228911174767
$ 1521 spim -f practice_q11.s
2*100000000000000000000000000000000000000000000000000000000000000000000000000+3*4
2000000000000000000000000000000000000000000000000000000000000000000000000012
```

NOTE:

No error checking is required.

Your program can assume it is given *exactly* one line of input.

Your program can assume this line is made up of only these 12 characters: `0123456789+*`

You can assume the line contains less than 10,000 characters.

Your program can assume that these characters form a valid arithmetic expression.

Your solution must be in MIPS assembler only.

You are not permitted to use floating point operations or registers.

When you think your program is working, you can run some simple automated tests:

```
$ 1521 autotest practice_q11
```

When you are finished working on this activity you must submit your work by running give:

```
$ give cs1521 practice_q11 practice_q11.s
```

To verify your submissions for this activity:

```
$ 1521 classrun -check practice_q11
```

Sample solution for `practice_q11.s`

```
# no sample solution will be provided for this question
```

Submission

When you are finished working on a question, submit your work by running **give**.

You can run **give** multiple times. Only your last submission will be marked.

Don't submit any questions you haven't attempted.

Do not leave it to the deadline to submit your answers. Submit each question when you finish working on it. Running autotests does not automatically submit your code.

You can check if you have made a submission with **`1521 classrun -check practice_q<N>`**:

```
$ 1521 classrun -check practice_q1
$ 1521 classrun -check practice_q2
...
$ 1521 classrun -check practice_q11
```

Remember you have until **Saturday 28 November 16:00** Sydney time to complete this exam (not including any extra time provided by ELS conditions).

Do your own testing as well as running **autotest**