Introduction to C language (234126)
Moed A, Spring semester 2018
July 9th, 2018

Staff: Dr. Yechiel Kimchi, Mr. Natan Yellin, Mr. Assaf Shomer

Exam Duration: 3hrs
Assisting material: None whatsoever (written, printed, electronic, wire, divine)

Instructions:

• Fill clearly your ID and other items on the front page using pen (black/blue) only.
• You are not allowed to use any external material except pen and brain.
• Never define main(). You are not allowed to use any library function, except I/O functions,
  o Unless explicitly permitted (if you need one, implement it yourself).
• There are 4 problems, 20 pages (a few extra, empty pages are at the end).
• Read the problem descriptions carefully, and follow the instructions.
• Write your solutions on this exam form only, at the available space. Much space does not
  mean long solutions – all solutions can be rather short.
• Clearly mark your solution, and designate your draft/scratch parts so we don’t grade them.
• You are advised to write your solutions in pen too (black/blue only), but pencils are OK.
• Write your solution clearly – if we cannot understand what you have written, it is wrong
  o You’ll be able to appeal later, if you think it was correct.
• You can implement additional helper functions as you wish, unless instructed otherwise.
• Don’t use global or static variables, nor should you use include/define commands.
• You may use a function from another section in solving a section (no circular dependency)
  – even if you have not solved the other section. Order among functions is irrelevant.
• You don’t have to check input sanity, unless explicitly instructed to do so.
• Solution’s complexity should be reasonable, but is not checked, unless explicitly required.
• You may add verbal explanations about your solution, but it is not considered for grading.
• The grade is for the code you write – not for your intentions.
  o There is no penalty for minor syntax errors that do not affect the algorithm.

We Wish You Success and Good Luck
Problem 1 (20 points)
Here is a complete program. Fill in the output of the program at the marked locations.

```c
#include <stdio.h>
int y = 10;        /* A global variable */
void f4(int *w)
{ *w = 1;
}
void f3(int *x)
{ *x = 2;
}
void f2(int x, int *z)
{ f3(&x);
  f4(&y);
  *z = 3;
  printf("x=%d y=%d z=%d\n", x, y, *z);
}
void f1(int x, int z)
{ x = 10;
  y = 11;
  z = 12;
  printf("x=%d y=%d z=%d\n", x, y, z);
}
int main(void)
{ int x = 9;
  int y = 9;
  printf("x=%d y=%d\n", x, y);
  /* output: x =_______  y =________ */
  f1(x,y);  /* output: x =_______  y =________  z =________ */
  printf("x=%d y=%d\n", x, y);
  /* output: x =_______  y =________ */
  f2(x,&y); /* output: x =_______  y =________  z =_____ */
  return 0;
}
```

x=%d y=%d

- 3 -
Problem 2 (20 points)

In this problem the solution must be recursive: No loops are allowed whatsoever.

The function int convert(char *s) is given a string that contains an integral number via the parameter s and it returns the value of that number as an int.

There are a couple of twists in the problem:

1. The string may contain spaces (only simple space: ’ ’) preceding the number, or after it.
2. The number may be preceded by a sign: – (minus) or + (plus), that affects the value. It is guaranteed that there is at most one sign (no sign is treated as + sign), and no spaces appear between the sign (if exists) and the digits.
3. The number ends when there is a space or ’\0’ after the digits. No further analysis is required when the end of the number is found (Example: If the string is “ –12 zoo”, the result is –12, and so is for the string “ –12 000”.

Use a reasonably efficient algorithm. Remember: The solution must be recursive – no loops!

int convert(char *s) /* See next page */
char *eat_space(char *s)
{
    if (*s != ' ') return s;
    return eat_space(s+1);
}

int evaluate(char *s, int head)
{
    int num, val;
    if (*s == ' ' || *s == '\0') return head;
    val = head * 10 + *s - '0';
    num = evaluate(s+1, val);
    return num;
}

int convert(char *s)
{
    int sig = 1;
    s = eat_space(s);
    if (*s == '-') {
        sig = -1;
        ++s;
    }
    if (*s == '+') ++s;
    return sig * evaluate(s, 0);
}
**Problem 3 (30 points)**

In this problem you will implement a basic compression algorithm using the counts of repeated characters. For example, the string “aaaaxbceccaa” will be compressed to the string “4axbb3c3a”. The idea is that repeated characters are compressed to a single representative that is preceded by the multiplicity of that character. Exceptional cases are multiplicities < 3: There is no point in replacing “bb” by “2b”, and even worse is replacing “b” by “1b”. The input string is guaranteed to contain only uppercase and lowercase letters. However, the two cases are considered equal; namely, the string “AaAaXBbYYY” will be compressed to “4axbb3y” (i.e., the result contains no uppercase letters). The function `double compress(char *s)` compresses the string that is given by `s` in-place; namely, the compressed string replaces the given string (and is, usually, shorter). You must add a ‘\0’ character at the end of the compressed string. The function returns the ratio of the compression; namely, the floating point value of the new length divided by the old length. Example: For the 2nd example above the function will return `7/10`; namely, 0.7.

Implement all the required parts of the code – no usage of library functions is allowed.

**Your algorithms should be reasonably efficient.**

**Note:** If your solution misses/errs only in evaluating the compression ratio, but fully implements the compression part with no errors, you are guaranteed to get at least 25pts.

```c
double compress(char *s) /* Not limited to letters! */
{
    int fill_num(char *p, int num)
    {
        int d;

        if (num == 0) return 0;

        d = fill_num(p, num/10);

        *(p + d) = '0' + num % 10;

        return d + 1;
    }

    int fill(int count, int c, char *p)
    {
        int digs = fill_num(p, count);

        *(p+digs) = c;

        return digs + 1;
    }
}
```
int to_lower(int c)
{
    if (c < 'A' || 'Z' < c) return c;
    return c - 'A' + 'a';
}

double compress(char *s) /* Not limited to letters! */
{
    int i = 0, j = 0, c = to_lower(s[i++]), count = 1;
    while (c != '\0') {
        while (c == to_lower(s[i])) {
            ++count; ++i;
        }
        switch (count) {
            case 2: s[j++] = c;
            case 1: s[j++] = c;
                    break;
            default: j += fill(count, c, s+j); /* updates j */
        }
        c = to_lower(s[i++]); count = 1;
    }
    s[j] = '\0';
    return (double)j/(i-1);
}
**Problem 4 (30 points)**

A table $m$ (2-dimensional array) is called right-0-triangular if:

1. In each and every row, if it contains 0s, they are all placed to the right of non-0s, if non-0 elements exist (see row $m[0]$).
2. For $i \geq 0$, $m[i+1]$ contains no more 0s (i.e., less equal many 0s) then $m[i]$.

See the example on the right hand side.

In this problem you have to implement the function `int Min0sAbove(int m[][M], int rows, int nz, int *found)`:

The function finds a row that contains a certain number of 0s, as hereby described. The function takes four parameters: an array $m[][M]$ (a table of size $rows*M$, where $M$ is #defined), the number of $rows$, the number $nz$ of the desired 0s, a pointer $found$ through which it reports the actual number of 0s that is found. The function searches the table $m[][M]$ for a row that has at least the number $nz$ of 0s and, if the exact number does not exist, it is the minimal number of 0s above $nz$. The function returns the index of the row which is best match.

**Examples:** For the table above, of size 10*14, if $nz$ is 0, the function returns 9, and 0 is sent via $found$ because $m[9]$ contains no 0s. If $nz$ is 7, the function returns 4, and sends 8 via $found$, because $m[4]$ has eight 0s, and it is the minimal number of 0s above 7, that any row has. If $nz$ is 15, then the function returns -1, because no row has 15 or more 0s, and $found$ is not used to update an external variable. **Use the most efficient algorithm that you can.**

```c
int Min0sAbove(int m[][M], int rows, int nz, int *found)
{
    int col = M - nz,
    row = col < 0 ? -1 : find_row(m, rows, col),
    j = row < 0 ? -1 : left_0(m[row], col);

    if (row < 0) return -1;

    *found = M - j;

    return row;
}
/* Complexity is O(log(rows) + log(M)) */
```
int find_row(int m[][M], int rows, int col)
{
    int b = 0, t = rows; /* bottom = 0, top = impossible */

    if (col >= M) return rows - 1;

    while (b < t - 1) {
        int mid = (b + t)/2;
        if (m[mid][col] == 0) {
            b = mid;
        } else {
            t = mid;
        }
    }

    return m[b][col] == 0 ? b : -1;
}
```c
int left_0(int r[], int col)
{
    int b = -1, t = col; /* bottom = impossible, top = possible */
    if (t >= M) t = M-1; /* top = possible - not guaranteed */
    while (b < t - 1) {
        int mid = (b + t)/2;
        if (r[mid] == 0) {
            t = mid;
        } else {
            b = mid;
        }
    }
    return r[t] == 0 ? t : M;
}
```
/* A correct implementation of the following algorithm, 
of complexity O(rows + M), gains 20 points */

```c
int Min0sAbove(int m[][M], int rows, int nz, int *found) {
    int i = rows, j = M - nz, rfound;

    if (j < 0) return -1;

    if (j < M) {
        for (i = 0; i < rows; ++i) {
            if (m[i][j] != 0) break;
        }
    } else {
        j = M - 1;
    }

    rfound = i - 1;

    if (rfound < 0) return rfound;

    for ( ; j >= 0; --j) {
        if (m[rfound][j] != 0) break;  }

    *found = M - j - 1;

    return rfound;
}
```