Series 3: C Programming - Pointers 2

Reminder

Have you read, in week 1 exercises series, the *advice* related to these series of exercises?

Exercise 1: Automatic letter generator (functions and character strings, level 1)

The goal of this exercise is to write a program named `letter.c`, which will constitute a (very simplistic) automatic generator of letters.

1. First, write a function `generateLetter()` (without arguments, and without return value). This function should simply produce the following output on the screen: (without the '`{`' and '`}`' characters.)

   ```
   Dear {Ms.} {Mireille},
   I am writing to you regarding {your course}.
   We should see each other on {12}/{18} to discuss it.
   Let me know!
   {Kind regards}, {John}.
   ```

   Simply call the `generateLetter()` function from the `main()` function. Compile your program and make sure it properly works.

2. Now modify the `generateLetter()` function, so as to make the parts in the preceding text between the curly bracket ("`{}`") configurable. To do this, you will need to pass a certain number of arguments to the function, in order to specify:

   - The appropriate name prefix for the introduction ("Ms." for a female recipient, and "Mr." for a male recipient); This choice must be made based on the value of an enumerated argument (e.g. `gender`);
   - The name of the recipient, named for example `recipient`;
   - The subject of the letter (parameter named `subject`);
   - The appointment date as two integer parameters, one for the day and one for the month;
   - The polite formula (`polite` parameter);
   - and the name of the author (`author`).

   The function will therefore have seven parameters.

   Invoke the function at least twice in a row from `main()`, setting the calls to produce the previous letter, and the response below:

   ```
   Dear Mr. John,
   I am writing to you regarding your meeting request.
   We should see each other on 12/16 to discuss it.
   Let me know!
   Sincerely, Mireille.
   ```

Exercise 2: Word segmentation (character strings, level 2)

In the `token.c` file, prototype then define the function:

```
int nextToken(char const * str, size_t* from, size_t* len)
```

whose role will be to find the position and length of the first word in the string `str` that comes after the position `from`.

Note: words are separated by a sequence of at least one space character (i.e. `' '`).

The function should update the `from` and `len` arguments to the index of the first character of the word and its length, provided such a word exists. If the word exists, the function should return a strictly positive value. Otherwise, the function should return `0`, and the updated values of `from` and `len` becomes meaningless.

From the `main` of the program, you will ask the user to enter a string of characters on the keyboard, and display (by making successive calls to the `nextToken()` function) all the words of the entered string. Words should be displayed per line and placed between apostrophes.

Use the operating example below to check the validity of your program; pay particular attention to ensure that the apostrophes surround the words **without there being a space in between**.

Also check that the program behaves correctly even when the entered string ends with a series of spaces.

Note: to read [an entire line], use:

```
fgets(line_to_read, size, stdin);
```

where `line_to_read` is an array of at least `size char`.

Since the sentence (read in `line_to_read`) is entered by the user, you cannot know the exact value that `size` should take. In this case, pre-define a size `MAX_SIZE` large enough for the user to enter a sentence of reasonable length.

Example of operation:

```
Enter a string: uh hello, this is my string!
The words of "uh hello, here is my channel!" are:
'uh'
'hello,'
'this'
'is'
'my'
'string'
'!'
```

Exercise 3: Integrals revisited (arrays, typedef, pointers to functions, level 3)

Pointers to functions

The aim here is to extend exercise 4 of series 1 in the direction suggested by the last remark (and as seen in class).

Copy your program `integrale.c` (or whatever name you gave it) to a new file, then modify it according to the description below.

Try (obviously!) to do this exercise **without** copying the corresponding slides from the course...

Define three functions of your choice that combine mathematical functions defined in `math.h` (`man sin`, `man exp`, `man sqrt`).

All these functions must be compatible with the prototype: `double f(double);`.

Define the type `Function` as a **pointer** to a prototype function similar to the one shown above.

Write a function `request_function()` that takes no arguments and returns a `Function`. This function must ask the user for a number between 1 and 5. If the user answers 1, 2 or 3, the `request_function()` should return a pointer to the corresponding function of yours. If he/she answers 4, it should return a pointer to the sine function (`sin`) and if he/she answers 5 it should return a pointer to the exponential function (`exp`).

Next calculate the integral of the returned function using the function `integrate()` (defined in exercise 4 of series 1) which you will have modified so that it takes as an additional argument: `Function`.

Arrays of pointers to functions

If you haven't already done it like this, we will now further refine the choice by implementing an array of `Function`s representing all the possible choices.

You can choose to implement it using a static array (define the size as a constant) or dynamically (i.e., pointer here [so you will have a pointer to a pointer to a function!]).

Define an array of `Function`s named `choice`.

The basic idea is to choose the function directly by indexing the array of possible choices. If the user's response is stored, for example, in the variable `rep`, then the correct function will be `choice[rep]`. (The function `request_function()` now returns an integer).

Initialize this array to match the previous choices (`f1, f2, f3, sin, exp`).

Test your program.

Finally, if you still have the courage and want to further refine your program, you can create a structure which contains the `Function` and a character string describing the function.

When asking the user, display the description of the function to help the user in their choice.

Execution example

```
You can choose from the following functions::
   1- x square
   2- root of exponential
   3- log(1+sin(x))
   4- sine
   5- exponential
Which function [1-5]  do you want to calculate the integral of? 4
Enter a real number: -3.141592653589
Enter a real number: 0
Sine integral between -3.14159265359 and 0:
-2.00001781364
```

Exercise 4: mini VM (pointers to functions, level 1)

Let's come back to function pointers by simulating a very simple command interpreter.

A command interpreter is simply an infinite (or almost) loop that matches an action (function call) to a string entered by the user.

We are going to make a very simple version here to illustrate the function pointers. Assume that we are working with a machine with 2 registers and 5 commands:

- `quit`, which simply displays "Bye!";
- `pop`, which writes the value of the 2nd register to the 1st;
- `push`, which writes the value of the 1st register into the 2nd, asks the user for a value (`double`), and puts it into the 1st register.
- `add`, which adds the value of the 2nd register to the 1st.
- `print` which displays the value of the 1st register.

If an unknown command is entered, it will be considered as `quit`.

Execution example

```
Enter a command (print, add, push, pop, quit) : print
-> 0
Enter a command (print, add, push, pop, quit) : push
Value ? 1.2
Enter a command (print, add, push, pop, quit) : print
-> 1.2
Enter a command (print, add, push, pop, quit) : push
Value ? 2.3
Enter a command (print, add, push, pop, quit) : print
-> 2.3
Enter a command (print, add, push, pop, quit) : add
Enter a command (print, add, push, pop, quit) : print
-> 3.5
Enter a command (print, add, push, pop, quit) : quit
Bye!
```

To work in a generic way (we might want to generalize a little more), all these commands will have a prototype:

```
void command(void* data);
```

and will interpret `data` as necessary: either one or two consecutive `double`s.

Write an `interpreter()` function that takes a `const char*` and returns a pointer to a command. This function must of course translate the name of a command into the corresponding function.

Finally finish your program by declaring in the `main` an array of two `double` initialized at 0 (which will serve as a representation of the 2-register memory of our machine) and by making a loop which requests a command from the user then execute this command. The loop will stop as soon as the command received is `exit` (see example of execution above).

Exercise 5: linked list (data structures, pointers, level 2)

We want to write the program `chainlist.c` which implements a version of linked lists.

1. Let's decide to implement a dynamic linked list of elements of type `type_el`.

   To make the thing easily editable, define this type, for example as an integer type.

2. Then define the type `ChainList`, representing a "linked list", based on the observation: a linked list is a sequence of elements each containing a value and the linked list of subsequent values.

3. Prototype and define the manipulation functions associated with lists:

   - inserting an element in the list after a given element,
   - inserting an element at the top of the list,
   - removing the first element from the list,
   - deleting a given element of the list (do not forget to **free** the memory requested during insertion for the element which is deleted)
   - and calculating the length of the list (number of elements) (for change, try coding it with a `for` loop).

   Improvement (Level 3): When calculating list length, be careful of cyclical lists (where the next of an element is a previous element).

Exercise 6: return to the memory explorer (pointer arithmetic; level 1)

Repeat exercise 1 of series 2, but write it using pointer arithmetic, for example displaying all addresses directly (without using an index):

```
0x7fffb7ed88ac : 01010000  80 ('P')
0x7fffb7ed88ad : 00000000   0
0x7fffb7ed88ae : 00000000   0
0x7fffb7ed88af : 00000000   0
```

In the same spirit, you can go back to other old exercises and re-write the solutions using pointer arithmetic. For example, exercise 2 above (word segmentation).

(more on next page, if you want)

Do you want more?

[optional] Exercise 7: MCQ questionnaire (structures + character strings + dynamic allocation, level 2)

We are trying here to create an examination program in the form of a multiple choice questionnaire (MCQ) where a question is asked and the answer is to be chosen from a set of proposed responses (only one correct response possible per question).

In a `mcq.c` program, define a `MCQ` structure including 3 fields:

1. a `question` field, of type character string, which will contain the question to ask;
2. a `responses` field which will be an array of at most 10 character strings containing the proposed responses;
3. a `nb_rep` field, of unsigned integer type, containing the number of possible responses;
4. an integer `solution` field (positive integer) which contains the number of the correct answer (in the `responses` field).

Prototype then define a `display()` function which takes a `MCQ` as an argument and displays it. For example :

```
How many teeth does an adult elephant have?
    1- 32
    2- from 6 to 10
    3- a lot
    4- 24
    5- 2
```

In the `main`, create and initialize the `MCQ` above, then display it. Compile and verify that everything works correctly so far.

Write a function `ask_number()` with 2 arguments, min and max, to ask for an integer in this range, and create a function `ask_question()` which takes a `MCQ` as an argument, successively calls `display` and `ask_number()` and returns the user's response.

Before continuing, test your program (displaying the question and entering the answer).

We now seek to examine several questions. Define type `Exam` as a dynamic set (pointer) of `QMC`. Create an `Exam` in the `main`, then fill it (in a `create_examination()` function is better!) with the following questions (*partial* code to complete) where `back` is a pointer to an `Exam`:

```
strcpy((*back)[0].question,
       "How many teeth does an adult elephant have");
(*back)[0].nb_rep = 5;
strcpy((*back)[0].reponses[0], "32");
strcpy((*back)[0].reponses[1], "from 6 to 10");
strcpy((*back)[0].reponses[2], "a lot");
strcpy((*back)[0].reponses[3], "24");
strcpy((*back)[0].reponses[4], "2");
(*back)[0].solution = 2;

strcpy((*back)[1].question,
       "Which of the following statements is a function prototype");
(*back)[1].nb_rep = 4;
strcpy((*back)[1].reponses[0], "int f(0);");
strcpy((*back)[1].reponses[1], "int f(int 0);");
strcpy((*back)[1].reponses[2], "int f(int i);");
strcpy((*back)[1].reponses[3], "int f(i);");
(*back)[1].solution = 3;

strcpy((*back)[2].question,
       "Who asks stupid questions");
(*back)[2].nb_rep = 7;
strcpy((*back)[2].reponses[0], "the math prof.");
strcpy((*back)[2].reponses[1], "my boyfriend/girlfriend");
strcpy((*back)[2].reponses[2], "the physics prof.");
strcpy((*back)[2].reponses[3], "me");
strcpy((*back)[2].reponses[4], "the programming prof.");
strcpy((*back)[2].reponses[5], "no one, there are no stupid questions");
```

```
strcpy((*back)[2].reponses[6], "polls");
(*back)[2].solution = 6;
```

Important: in this step, you will be required to make dynamic allocations. Remember to deallocate the memory once the work is finished (for example with a `destroy_examination` function).

To finish:

1. Ask the questions one by one;
2. Count the number of correct responses;
3. Give the score at the end.

Execution example

```
How many teeth does an adult elephant have?
    1- 32
    2- from 6 to 10
    3- a lot
    4- 24
    5- 2
Enter an integer between 1 and 5: 2
Which of the following statements is a function prototype
    1- int f(0);
    2- int f(int 0);
    3- int f(int i);
    4- int f(i);
Enter an integer between 1 and 4: 3
Who asks stupid questions
    1- the math prof.
    2- my boyfriend/girlfriend
    3- the physics prof.
    4- me
    5- the programming prof.
    6- no one, there are no stupid questions
    7- polls
Enter an integer between 1 and 7: 5
You found 2 correct answers out of 3.
```

[optional] Exercise 8: Multiple choice questions revisited (level 2)

This exercise repeats the previous exercise. We now want this program `mcq` to be independent of a particular MCQ (separation of the algorithm and the data).

For this we will give this program the possibility of reading the MCQs in a file.

Copy the `mcq.c` program into a new file then edit it.

Start by copying the `request_file()` function from exercise 13 of series 1 (`stat.c`).

Then change the `create_examination()` function from the previous exercise (the one which created the MCQ) so that it creates the content of the MCQ from a file passed as an argument:

```
Exam create_examination(FILE* file);
```

The format of the file to read is as follows:

```
Q: question
reponse1
->reponse2
reponse3

Q: question2
...
```

The `->` sign at the start of a line indicates the correct response. The `#` sign at the start of a line indicates a comment line.

If the response itself starts with "`->`" or "`#`", avoid placing that sign right at the beginning of the line, but instead insert a space. The reading procedure should eliminate these initial spaces.

Note: the advantage of this format is that the order of the responses can easily be changed in the configuration file without having to modify the indication of the correct response.

Example file:

```
Q: How many teeth does an adult elephant have?
32
-> from 6 to 10
a lot
24
2

Q: Which sign is the strangest?
 #
 ->
->->##<-
#b
a
```

The last question corresponds to

```
Which sign is the strangest?
    1- #
    2- ->
    3- ->##<-
    4- a
```

and the answer is `3`.

Create the file corresponding to last time's questionnaire in `exam1.txt` and test your program.

[optional] Exercise 9: Address book (pointers, files, character strings; level 3)

[proposed by T. Coppey, 2010]

The goal of this exercise is to create a complete mini-application that includes a data structure (unbalanced binary trees), files (read/write) and a small command interpreter.

A binary tree allows you to store different objects ordered by a key. In our address book, we will use the person's name as key and the telephone number as values (but you can of course add other fields).

1. Create an `abook.c` file where you can copy into the prototypes below. You will also complete the `addr__` structure which represents a node of the tree so that it contains (in addition to the tree structure): the name, the telephone number, and possibly other fields (address, e-mail, ...).

```c
/* a node of the tree */
typedef struct addr__ addr_t;
struct addr__ {
    /* to be completed */
};

/* the tree, defined by its root */
typedef struct book__ {
    addr_t* root;
} book_t;

/* create a new empty address book */
book_t* book_create(void);

/* release associated resources */
void book_free(book_t* b);

/* list in order all the names in the address book */
void book_list(book_t* b);
```

```c
/* display an address book entry */
void book_view(book_t* b, const char* name);

/* add or modify an address book entry */
void book_add(book_t* b, const char* name, const char* num);

/* delete an entry from the address book */
void book_remove(book_t* b, const char* name);

/* replace the contents of the address book with those of the file */
void book_load(book_t* b, const char* file);

/* save the contents of the notebook in a CSV format file */
void book_save(book_t* b, const char* file);
```

2. Then implement the functions that have been prototyped (you are free to modify the prototype if it suits you). The functions are listed somewhat in ascending order of difficulty.

   Note: the CSV format is: one name-number record per line, where the fields are separated by ';'. For example:

   ```
   Lucien;012 345 67 89;
   Stéphane;021 879 51 32;
   Julien;079 523 12 45;
   Antoine;076 125 08 78;
   Damien;022 329 08 85;
   ```

3. To be able to manipulate your address book you will need a command interpreter. The mode of operation is very simple, you must

   - read the line that contains the command
   - split the received string into arguments
   - depending on the first argument, call the corresponding function with the correct arguments
   - restart

   To split the received string into arguments, you can use the following code:

   ```c
   /* we divide the command into arguments, see man strsep */
   int  an=0;  /* number of arguments  */
   char *a[10];   /* arguments array */
   char *ptr=cmd;  /* cmd is the buffer that contains the line */
   while (an<10 && (a[an]=strsep(&ptr," \t\n"))!=NULL) {
       if (a[an][0]!='\0') ++an;
   }
   /* arguments are in a[0] .. a[an-1] */
   ```

   The interpretations of the commands are as follows. They correspond directly to a function that you have previously implemented.

   ```
   add <name> <num>  add a number
   del <name>        delete a number
   view <name>       show information
   list              list names
   load <file>       read file addresses
   save <file>       save addresses to file
   quit              quit program
   ```

The bravest among you will go even further by implementing the command interpreter in the form of an array of pointers to generic functions.

[optional] Exercise 10: Needleman-Wunsch algorithms (Viterbi algorithm on the editing graph) and Longest Common Subsequence (level 2)

[adapted by T. Coppey (2010) based on an old exam subject.]

The comparison and alignment of character sequences is a fundamental task in several fields of application, notably in speech recognition, image processing and bioinformatics.

The aim of this exercise is to implement the Needleman-Wunsch algorithm (which is a Viterbi algorithm), used for example in the BioWall that you can in Musée Bolo, not far from the IN exercise rooms.

The first part of the algorithm consists of evaluating the comparison between two character strings of different sizes, by minimizing the number of insertions of "holes" between two characters of the same string. In practice, a matrix `M` of size `(m + 1) × (n + 1)` is constructed, where m and n are the numbers of characters of the two strings to be compared. A "hole" is characterized by a horizontal or vertical movement in the matrix. We then assign to each cell $M_{i,j}$ of the matrix a score defined by the following equations:

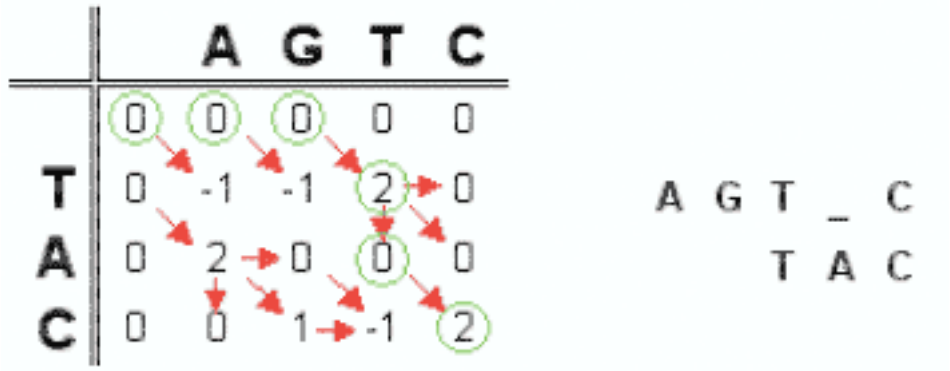
$$\begin{aligned} M_{i,0} = M_{0,j} &= 0 && \text{for } 0 \leq i \leq m \text{ and } 0 \leq j \leq n \\ M_{i,j} &= \max\{\, M_{i-1,j-1} + S_{i,j},\ M_{i,j-1} - 2,\ M_{i-1,j} - 2 \,\} && \text{for } 1 \leq i \leq m \text{ and } 1 \leq j \leq n \end{aligned}$$

with

$$S_{i,j} = \begin{cases} 2 & \text{if } A_i = B_j \\ -1 & \text{if } A_i \neq B_j \end{cases}$$

where A_i designates the i$^{\text{th}}$ character of the first string and B_j the j$^{\text{th}}$ character of the 2nd string.

The goal is to implement in C a function that creates this matrix from two character strings entered by the user. Each cell must keep, in addition to its score, the one of the two parent cell(s), for which the maximum has been reached in the above equation. Here is an illustration where red arrows represent parent cells:



The second part of the algorithm finds a correct alignment. To do this, you must search for a path starting from cell $M_{m,n}$ and arriving at cell $M_{0,0}$, each time using the stored parent cell.

To implement this algorithm, proceed in the following steps:

1. In the `needle.h` file describe the data structures:
   - To compare 2 strings `s1` and `s2` of respective lengths `l1` and `l2`, you will need to create an array of `(l2 + 1) x (l1 + 1)` cells (by placing s1 horizontally and s2 vertically).
   - Each cell contains a (numeric) value and a reference to the predecessor cell (there are only 3 possibilities: immediately left, top or diagonal).
2. Then write in `needle.c` a function `computeTable()` which takes the two character strings as input and constructs the array by assigning to each cell a predecessor and a value according to the previous equations:
   - For the first line, the predecessor is on the left and the value 0.
   - For the first column, the predecessor is at the top and the value 0.
   - Then fill in the array according to the previous equations, memorizing the predecessor
3. Write a function `extractSolution()` that takes the table as input and calculates the path from $M_{0,0}$ to $M_{m,n}$ using the predecessor of each cell (Hint: you will have to start calculating the path from the end and then reverse it).
4. You can then test your program using the function

```
void showSolution(Solution* sol, const char* s1, const char* s2) {
    int i,j;
    for (i=0,j=0; i<sol->size; ++i) {
        switch (sol->dirs[i]) {
            case DirDiag: /*continued*/
            case DirHorz: printf("%c",s1[j]); ++j; break;
            case DirVert: printf("_");
```

```
        }
    }
    printf("\n");
    for (i=0,j=0; i<sol->size; ++i) {
        switch (sol->dirs[i]) {
            case DirDiag: /*continued*/
            case DirVert: printf("%c",s2[j]); ++j; break;
            case DirHorz: printf("_");
        }
    }
    printf("\n");
}
```

Which will give you for the values:

```
s1 = "Hello sir, what time is it on your watch ?"
s2 = "Have a good day madam, may the happy little girl show you the way"
```

the following result:

```
_____________________Hello sir, what__ tim__e _is_ it on your watch_ ?__
Have a good day madam, may the_ _happy little girl sh_ow you_ __t_he way
```

5. **Extension**: You can now easily replace the Needleman-Wunsch algorithm with the Longest Common Subsequence (LCS) algorithm if you wish; simply replace the formula with:

$$M_{i,j} = \begin{cases} M_{i-1,j-1} + 1 & \text{if } A_i = B_j \\ \max\{\ M_{i,j-1},\ M_{i-1,j}\ \} & \text{if } A_i \neq B_j \end{cases}$$

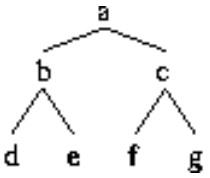
[optional] Exercise 11: Formal derivation (level 3)

[adapted by T. Coppey (2010) based on an old exam subject.]

The goal here is to make a program capable of formally deriving expressions.

For this we will use a **binary tree** structure. A binary tree is a data structure defined recursively by: A tree that is

- either a value;
- or a value and two (binary) subtrees.



For example, if we denote a tree in parentheses by first indicating its right subtree, then its value, and finally its left subtree, then:

```
(((d) b (e)) a ((f) c (g)))
```

is a binary tree: with value `a` and two subtrees `((d) b (e))` et `((f) c (g))`.

Similarly `(f)` is a one-valued tree (i.e. has no subtree).

1. Based on your knowledge of linked lists, define a data structure to represent binary trees.

   In this data structure, the value will be represented by a `char`.

2. Then write the function allowing you to create a binary tree from a value and two subtrees.

3. Write a function to display a binary tree in parenthesized format as illustrated above.

   We will recursively display the left subtree, then the value then (recursively) the right subtree.

4. In the `main()`, test that your code works correctly.

   Optional: make sure that the same tree can share one or more subtree(s). For example, the tree `((x) + (x))` can be *constructed* from **a single** tree `(x)`. To do this, you will either use deep copy or a reference counter.

5. Now let's move on to the representation of arithmetic expressions.

   We will use binary trees for this (we will only consider here the binary operators `+ - / *` and `^` (power)):
   for example `a + b` will be represented by the tree `((a) + (b))` where '`+`' is the value of the first tree and "`(a)`" and "`(b)`" are the subtrees.
   Likewise the arithmetic expression `(a + b) * (c + d)` will be represented by the tree `(((a) + (b)) * ((c) + (d)))`.

   Construct trees representing the following expressions (4 expressions): x + a (x + a) * (x + b) ((x * x) * x) + (a * x) (x ^ a) / ((x * x) + (b * x))

   Hint: start by creating the binary trees representing the expressions `a`, `b` and `x` (alone), then those representing the expressions `x+a`, `x+b` and `x* x`.

6. Then write a function `derive()` which takes a binary tree as an argument, assumed to be a representation of a valid arithmetic expression and a character representing the variable with respect to which to derive, and which returns the binary tree corresponding to the derived arithmetic expression.

   We will proceed to do this recursively, remembering the usual derivation rules:

   ```
   da/dx = 0  where a is a constant (here: character different from x)

   dx/dx = 1

   d(f+g)/dx = df/dx + dg/dx

   d(f-g)/dx = df/dx - dg/dx

   d(f*g)/dx = (df/dx * g) + (f * dg/dx)

   d(f/g)/dx = ((df/dx * g) - (f * dg/dx) ) / (g * g)

   d(f^a)/dx = a * df/dx * (f ^ (a - 1))

   (where a does not depend on x, which we will assume here)
   ```

   Important note: we *do not* want the minimal derivation tree! Clearly, we will not seek to simplify the obtained expressions.

7. Test your differentiation function on the 4 previous expressions.