**ЗВІТ ПРО ВИКОНАНЕ ЗАВДАННЯ**

з курсу “Теорія програмування”

студента групи ПК-12-2

Бекленищева Владислава Ігоровича

ЗАВДАННЯ № 2

кафедра комп’ютерних технологій, ДНУ

2015/2016 навч. р.

# 

# Постановка задачи

Написати програми на мові OCAML для роз'язання вказаних нижче завдань.  
Продемонстрирувати їх роботу на тестових прикладах в середовищі OCAML.  
Підготувати звіт згідно до форми (див. файл Вимоги.TXT)

Вариант - 1

Слияние двух списков:

1. Написать функцию *merge\_i*, которая получает на входе два списка целых чисел, отсортированные в порядке возрастания, и возвращает новый отсортированный список содержащий элементы первых двух списков.
2. Написать обобщенную функцию *merge*, которая получает в качестве аргумента функцию сравнения и два списка, отсортированных в этом порядке, и возвращает объединённый список в том же порядке. Функция сравнения будет следующего типа: **‘a -> ‘a -> bool**.
3. Примените эту функцию к двум спискам целых чисел отсортированных в порядке не возрастания, затем к двум спискам строк отсортированных в порядке не возрастания.
4. Что случится если один из списков не будет в порядке не возрастания?
5. Напишите новый тип ***list*** в форме записи, содержащей три поля: обычный список, функция порядка и логическое значение, указывающее является ли список в том же порядке.
6. Напишите функцию *insert*, которая добавляет элемент в список этого типа.
7. Напишите функцию *sort*, которая сортирует вставками элементы списка.
8. Напишите новую функцию *merge* для этих списков.

# Описание решения

Условно всё задание можно поделить на две части:

1. В первой половине работы я создавал функции **merge\_i** и **merge** для обычных списков и тестировал работу на разных входных данных. Первая половина описана в пунктах “Задача 1” - “Задача 4”.
2. Во второй половине работы демонстрируется создание нового типа list\_t, который представляет из себя запись (record) содержащую 3 поля: list\_item - список, func - функция сравнения и order - порядок сортировки списка list\_item. Также здесь реализованы функции **insert**, **sort**, **merge** специально для записей типа list\_t. Вторая половина задачи описана в пунктах “Задача 5” - “Задача 8”.

## Задача 1

В данной задачи автор просит написать функцию **merge\_i**, которая получает два списка целых чисел отсортированных в неубывающем порядке. В результате своей работы функция возвращает новый список являющийся результатом слияния двух указанных списков. Порядок сортировки нового списка будет такой же как у исходных списков - неубывающий. На это указывает буква “i” в конце названия функции - **i**ncreased.

Основная идея алгоритма:   
На каждом уровне рекурсии сравниваем головы полученных списков и меньшую по значению (так как требуемый порядок неубывающий) добавляем в новый список. Если мы добавляем голову из первого списка, то передаём функции **merge\_i** хвост первого списка и второй список. Аналогично будет и со вторым списком. Главное условие выглядит следующим образом:

if head\_x <= head\_y then  
 head\_x::(merge\_i tail\_x ly)  
else head\_y::(merge\_i tail\_y lx)

В результате, на каждом уровне рекурсии у нас будет с помощью оператора “::” добавляться либо head\_x либо head\_y, а на самом верхнем уровне рекурсии у нас получиться комбинация всех элементов исходных списков образовывавшая новый список целых чисел, отсортированных в порядке неубывания (далее для невозрастающего порядка реализована функция **merge**).

## Задача 2

В центре задачи рассматривается обобщенная функция **merge** (а также функция **merge\_lists**), которая позволяет делать слияния двух списков с одинаковыми порядками сортировок. Функция принимает такие параметры:

1. compare\_func - функцию сравнения элементов. По условию функция должна быть следующего типа: **‘a -> ‘a -> bool**.
2. Два списка отсортированных в одинаковых порядках.

Функция сравнения выглядит следующим образом:

let compare a1 a2 =

if a1 <= a2 then true

else false

;;

Эта функция является общей для всех задач (1-8) и будет использоваться в дальнейшем.

Рассмотрим функцию **merge** подробнее. Вообще, основная идея та же, что и в прошлой задачи, но к ней добавляется еще и то, что списки могут быть отсортированы в невозрастающем порядке.

В начале своей работы функция определяет порядок сортировки элементов исходных массивов. В этом ей помогает вспомогательная функция **find\_list\_order**:

let rec find\_list\_order compare\_func l =

match l with

| [] -> true

| head::tail ->

match tail with

| [] -> true

| headnext::tailnext ->

if compare\_func head headnext then

find\_list\_order compare\_func tail

else false

;;

Если каждый следующий элемент больше предыдущего, то мы имеем список с неубывающим порядком сортировки и функция возвращает true. Если хотя бы один элемент меньше предыдущего, то список с невозрастающим порядком сортировки и функция возвращает false. Единственное условие для функции - список на входе должен быть отсортирован.

После определения порядков функция **merge** сравнивает их значения и если порядки не равны, то возникает исключение Different\_orders. Это исключение я создал специально для решения проблемы в задаче 4.

Если полученные порядки равны, то выполняется основная часть функции **merge** - вызов функции **merge\_lists**, которая отвечает за само слияние. Функция получает следующие параметры:

1. Функцию сравнения (compare\_func)
2. Порядок сортировки в качестве логического значения (sort\_order). sort\_order будет равен true если оба списка отсортированы в порядке неубывания, иначе - false.
3. Первый список (lx)
4. Второй список ly

Перед самим слиянием в функции **merge\_lists** проверяется порядок двух списков с помощью такого условия:  
if (sort\_order == true) then  
 if (compare\_func head\_x head\_y) then  
 head\_x::(merge\_lists compare\_func sort\_order tail\_x ly)  
 else head\_y::(merge\_lists compare\_func sort\_order lx tail\_y)  
else  
 if (compare\_func head\_x head\_y) then  
 head\_y::(merge\_lists compare\_func sort\_order lx tail\_y)  
 else head\_x::(merge\_lists compare\_func sort\_order tail\_x ly)

Функция **merge\_lists** принимает много входных параметров, но является довольно гибкой функцией в том плане, что она вызывается функциями **merge** и **merge\_structures** (так называется функция merge из задачи 8, позволяет выполнять слияние двух списков типа list\_t) с одной единственной целью - сделать слияния двух списков.

## Задача 3

По условию задачи 3, функцию **merge** нужно применить и к спискам целых чисел отсортированных в порядке не возрастания. Решения этой задачи реализовано в самой функции **merge**.

Функция **merge** работает также корректно и со списками строк, так как функция сравнения - **compare** работает с любыми типами данных.

Все примеры работы функций смотрите в разделе “Описание тестовых примеров”.

## Задача 4

Для списков, элементы которых отсортированы в разных порядках, функция **merge** работает некорректно (см. описание тестовых примеров). Поэтому я создал пользовательское исключение и вызываю его в случае разных порядков:

exception Different\_orders;;

if (order\_lx == order\_ly) then

merge\_lists compare\_func order\_lx lx ly

else raise Different\_orders

Таким образом, я просто освобождаюсь от ситуации с разными порядками списков (true и false или false и true). Дело в том, что при таком раскладе списки сливаются некорректно в том плане, что сначала учитывается сортировка первого списка, а потом уже второго.

Теперь перейдём ко второй части задания.

## Задача 5

В пятой задаче автор просит написать новый тип list в форме записи (record). Я его назвал **list\_t** и определил таким образом:

type 'a list\_t =

{

list\_item: 'a list;

func: 'a -> 'a -> bool;

order: bool;

}

В record типа list\_t мы храним:

1. список list\_item, который может быть отсортирован, а может быть и не отсортирован.
2. функцию сравнения func типа 'a -> 'a -> bool. Во всех случаях в качестве функции сравнения мы используем нашу функцию **compare**.
3. и порядок сортировки нашего списка list\_item. И здесь есть один нюанс. В первом пункте я сказал, что не важно отсортирован список или нет. Отсюда следует вопрос: если мы используем не отсортированный список и обязаны указать order, то какого значения должен быть order? Я предполагаю, две ситуации: либо список отсортирован и order указан верно, либо список не отсортирован и order указан неверно. Во втором случае может быть, например, список [1;7;4;3], а order равняться true (или false). Если пользователь в дальнейшем захочет отсортировать список, то он всё-равно будет вводить новое значение order в качестве параметра функции sort.

## Задача 6

Функция **insert** позволяет добавлять элемент x в список list\_item записи типа list\_t. При этом функция **insert** добавляет элемент таким образом, чтобы не нарушать порядок сортировки элементов списка. Стоит отметить, что функция **insert** выполняет меньшую работу - она только возвращает новую запись с новым списком list\_item. А в list\_item элемент х добавляется с помощью функции **insert\_item**. Функция определяется на верхнем уровнем. Я сделал это для того, чтобы дальше снова вызывать функцию **insert\_item**, но для других целей (для сортировки вставками - **sort**). Тем более insert\_item работает с элементами списков любого типа, что очень удобно.

Идея алгоритма состоит в том, чтобы проверять элемент х с головой списка list\_item или головой хвоста этого же списка. Если x меньше (больше - в случае невозрастающего порядка), то мы добавляем x к списку таким образом:   
x::(head::tail)   
А если x больше (меньше), то мы рекурсивно вызываем снова функцию **insert\_item** передавая ей порядок сортировки элементов текущего списка, хвост (tail) и сам элемент x:  
head::(insert\_item compare\_func sort\_order tail x)

## Задача 7

В этом пункте мы рассмотрим функцию sort, которая позволяет отсортировать список list\_item указанной записи list\_struct методом “instertion sort” (сортировкой вставками). Указанный порядок сортировки подаётся в виде sort\_order:

* true - если нужно отсортировать массив в неубывающем порядке;
* false - в невозрастающем порядке.

Внутри функции sort алгоритм реализован в виде рекурсивной функции **insertion\_sort**, которая получает на входе список l (на первом шаге - сам list\_item):  
let rec insertion\_sort l =

match l with

| [] -> []

| head::tail ->

insert\_item compare\_func sort\_order (insertion\_sort tail) head

in insertion\_sort list\_struct.list\_item;

Внутри **insertion\_sort** я вызываю ту самую ранее определённую **insert\_item**, которая позволяет в отсортированную последовательность вставлять элемент. Сам вызов довольно длинный, но это стоит того чтобы использовать функцию **insert\_item** в двух функциях.

Для наглядности распишу как работает функцию **insertion\_sort** на примере [2;4;3]:

* на нижнем уровне рекурсии **insertion\_sort** вернёт пустой список [], в который **insert\_item** и добавит head: [3].
* уровнем выше будет следующее: insert\_item compare true [3] 4. В данном случае получим: 3::4.
* и на самом верхнем уровне будет insert\_item compare true 3::4 2. В результате окончательно получим: 2::3::4.

## Задача 8

В данной задаче автор просит написать функцию merge для записей. То есть, практически тоже самое, что было в задаче 2, но только для списков list\_item. Функцию merge для записей я решил назвать **merge\_structures** дабы не путать с ранее определённой в пункте 2 функцией **merge**.

**merge\_structures** принимает 2 параметра - lstruct\_x и lstruct\_y, которые принадлежат типу list\_t. Функция возвращает новую запись типа list\_t вместе с:

1. комбинированным массивом list\_item;
2. функцией сравнения, записанной в func;
3. и порядком сортировки, записанным в order.

Должен сказать, что новые значения func и order получаются с помощью копирования соответствующих значений из записи lstruct\_x, то есть из первого списка list\_item. И в последующей функции **merge\_lists** везде используются поля из lstruct\_x.

Само слияние списков list\_item выполняет рекурсивная функция **merge\_lists**, которая на входе получает почти те же параметры, что и в первом случае: функцию сравнения (compare), требуемый порядок сортировки (берём из lstruct\_x.order), список lstruct\_x.list\_item и список lstruct\_y.list\_item. Возвращает функция новый список list\_item.

Функция **merge\_structures** является практически копией функции **merge** из первой части задания, но с адаптацией под записи типа list\_t. Здесь не требуется вызывать функцию **find\_list\_order** для определения порядка сортировки списка. Это значение задаётся в каждой из lstruct\_x и lstruct\_y в поле order. В этом смысле функция **merge\_structures** работает намного удобнее своей предшественницы. К недостатку обеих функций я отнесу порождение исключения при разных порядках сортировок:  
if (lstruct\_x.order == lstruct\_y.order) then  
(\* … \*)  
else raise Different\_orders

# Исходный текст программы решения задачи

(\*

\* [title]: Lab #2, variant-1, "Merging two lists"

\* [author]: Beklenyshchev Vladislav, PK-12-2, DNU

\*)

(\*\*\*\*\*\*\*\*\*\* !!general function and exception for both parts of work!! \*\*\*\*\*\*\*\*\*)

let compare a1 a2 =

if a1 <= a2 then true

else false

;;

(\* raised in merge and merge\_structures\*)

exception Different\_orders;;

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* FIRST part of work \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\*

\* [description]:

\* merge lists lx and ly in one list using increasing order

\* [arguments]:

\* lx - 1st list

\* ly - 2nd list

\* [return value]:

\* new list as a result of merged lx and ly

\*)

let rec merge\_i lx ly =

match lx with

| [] -> ly

| head\_x::tail\_x ->

match ly with

| [] -> lx

| head\_y::tail\_y ->

if head\_x <= head\_y then

head\_x::(merge\_i tail\_x ly)

else head\_y::(merge\_i tail\_y lx)

;;

(\*

\* [description]:

\* this function find order of list l

\* [arguments]:

\* compare\_func - function which compare values of type 'a

\* l - list

\* [return value]:

\* true -> increasing order

\* false -> decreasing order

[hint]:

\* function is called only by <merge> function!

\*)

let rec find\_list\_order compare\_func l =

match l with

| [] -> true

| head::tail ->

match tail with

| [] -> true

| headnext::tailnext ->

if compare\_func head headnext then

find\_list\_order compare\_func tail

else false

;;

(\*

\* [description]:

\* function merge lists lx and ly

\* [arguments]:

\* compare\_func - function which compare values of type 'a

\* sort\_order - required order of sorting with next values:

\* true -> increasing order

\* false -> decreasing order

\* lx - fst list

\* ly - snd list

\* [return value]:

\* new list as a result of merged lx and ly

\* [hint]:

\* function is called by <merge> and <merge\_structures> functions!

\*)

let rec merge\_lists compare\_func sort\_order lx ly =

match lx with

| [] -> ly

| head\_x::tail\_x ->

match ly with

| [] -> lx

| head\_y::tail\_y ->

if (sort\_order == true) then

if (compare\_func head\_x head\_y) then

head\_x::(merge\_lists compare\_func sort\_order tail\_x ly)

else head\_y::(merge\_lists compare\_func sort\_order lx tail\_y)

else

if (compare\_func head\_x head\_y) then

head\_y::(merge\_lists compare\_func sort\_order lx tail\_y)

else head\_x::(merge\_lists compare\_func sort\_order tail\_x ly)

;;

(\*

\* [description]:

\* function find order of lists lx and ly and merge them

\* [arguments]:

\* compare\_func - function for comparing items from lx and ly lists

\* lx - fst list

\* ly - snd list

\* [return value]:

\* new list as a result of merged lx and ly

\*)

let merge compare\_func lx ly =

let order\_lx = (find\_list\_order compare\_func lx)

and order\_ly = (find\_list\_order compare\_func ly) in

if (order\_lx == order\_ly) then

merge\_lists compare\_func order\_lx lx ly

else raise Different\_orders

;;

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* SECOND part of work \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

type 'a list\_t =

{

list\_item: 'a list;

func: 'a -> 'a -> bool;

order: bool;

};;

(\*

\* [description]:

\* function added x item to l 'a list

\* [arguments]:

\* compare\_func - function which compare values of type 'a

\* sort\_order - required order of sorting with next values:

\* true -> increasing order

\* false -> decreasing order

\* l - list for insertion

\* x - item

\* [return value]:

\* sorted record of type list\_t

\* [hint]:

\* function is called by <insert> and <sort> functions!

\*)

let rec insert\_item compare\_func sort\_order l x =

match l with

| [] -> [x]

| head::tail ->

if sort\_order then

(\* increasing order\*)

if compare\_func x head then

x::(head::tail)

else head::(insert\_item compare\_func sort\_order tail x)

else

(\* decreasing order\*)

if compare\_func x head then

head::(insert\_item compare\_func sort\_order tail x)

else x::(head::tail)

;;

(\*

\* [description]:

\* function added x item to list\_struct of type list\_t

\* [arguments]:

\* list\_struct - list record of type list\_t

\* x - item to add in list\_item of current list\_struct

\* [return value]:

\* new list\_struct of type list\_t with inserted x

\*)

let insert list\_struct x =

{

list\_item =

insert\_item list\_struct.func list\_struct.order list\_struct.list\_item x;

func = list\_struct.func;

order = list\_struct.order;

};;

(\*

\* [description]:

\* function sort current list\_t in specified order

\* [arguments]:

\* compare\_func - function which compare values of type 'a

\* sort\_order - required order of sorting with next values:

\* true -> increasing order

\* false -> decreasing order

\* list\_struct - record of type list\_t

\* [return value]:

\* sorted record of type list\_t

\*)

let sort compare\_func sort\_order list\_struct =

{

list\_item =

(

let rec insertion\_sort l =

match l with

| [] -> []

| head::tail ->

insert\_item compare\_func sort\_order (insertion\_sort tail) head

in insertion\_sort list\_struct.list\_item;

);

func = compare\_func;

order = sort\_order;

};;

(\*

\* [description]:

\* merge two list\_item lists in one new list\_item and formes

\* new record of type list\_t

\* [arguments]:

\* lstruct\_x - fst record of type list\_t

\* lstruct\_y - snd record of type list\_t

\* [return value]:

\* record of type list\_t

\*)

let merge\_structures lstruct\_x lstruct\_y =

{

list\_item =

if (lstruct\_x.order == lstruct\_y.order) then

merge\_lists lstruct\_x.func lstruct\_x.order lstruct\_x.list\_item lstruct\_y.list\_item

else raise Different\_orders

;

func = lstruct\_x.func;

order = lstruct\_x.order;

};;

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* TESTS for part #1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

let l1 = [1;2;5;8];;

let l2 = [4;6];;

let l3 = [7;4;1];;

let l4 = [8;6;4;3];;

let s1 = ["Vlad"; "Tanya"; "Anya"];;

let s2 = ["Vlad";"Ruslan";"Nastya"];;

let s3 = ["Vlad"; "Vaad"; "Anya"];;

let s4 = ["Vlad"; "Vanya";"Vaad"];;

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* TESTS for part #2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\*example for increasing order\*)

let list\_struct1 =

{

list\_item = [1;3;6];

func = compare;

order = true;

};;

let list\_struct2 =

{

list\_item = [2;8;9;10;56];

func = compare;

order = true;

};;

(\*example for decreasing order\*)

let list\_struct3 =

{

list\_item = [7;4;1];

func = compare;

order = false;

};;

let list\_struct4 =

{

list\_item = [100;29;17;3;1];

func = compare;

order = false;

};;

# Описание интерфейса (руководство пользователя)

Работать с программой стоит непосредственно из интерпретатора OCaml.

Для задач 1-4 код можно копировать из первой части (first part), а для задач 5-8 - из второй части (second part). Весь код представлен в файле lab2\_var1.ml.

В конце файла есть тестовые примеры для заданий из первой части и заданий из второй части:

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* TESTS for part #1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

let l1 = [1;2;5;8];;

let l2 = [4;6];;

let l3 = [7;4;1];;

let l4 = [8;6;4;3];;

let s1 = ["Vlad"; "Tanya"; "Anya"];;

let s2 = ["Vlad";"Ruslan";"Nastya"];;

let s3 = ["Vlad"; "Vaad"; "Anya"];;

let s4 = ["Vlad"; "Vanya";"Vaad"];;

(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* TESTS for part #2 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)

(\*example for increasing order\*)

let list\_struct1 =

{

list\_item = [1;3;6];

func = compare;

order = true;

};;

let list\_struct2 =

{

list\_item = [2;8;9;10;56];

func = compare;

order = true;

};;

(\*example for decreasing order\*)

let list\_struct3 =

{

list\_item = [7;4;1];

func = compare;

order = false;

};;

let list\_struct4 =

{

list\_item = [100;29;17;3;1];

func = compare;

order = false;

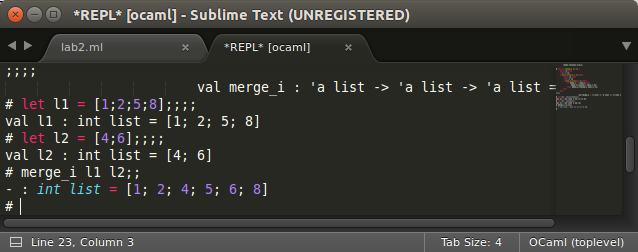
};;

Это должно немного ускорить демонстрацию лабораторной работы.

# Описание тестовых примеров

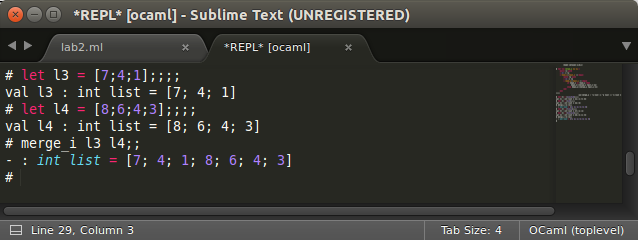
## Задача 1

Тест 1: “**merge\_i** для списка целых чисел отсортированного в порядке возрастания (increasing order)”



merge\_i позволяет сливать списки только в порядке возрастания.

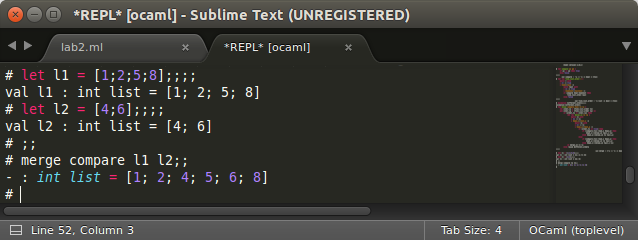
Тест 2: “**merge\_i** для списка целых отсортированного в порядке убывания (decreasing order)”



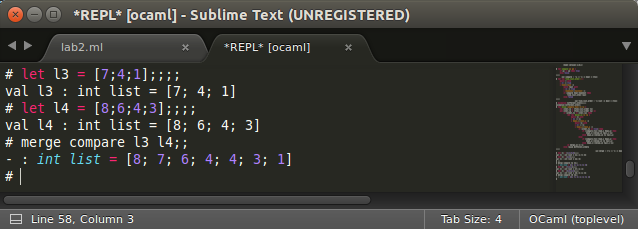
Так как **merge\_i** позволяет сортировать списки только в порядке возрастания, поэтому если мы передаём ей списки в разных порядках сортировки, то мы соответственно получаем неверный результат.

## Задача 2

Тест 1: “тестирование обобщенной функции **merge** с двумя списками отсортированных в порядке не убывания (оба)”



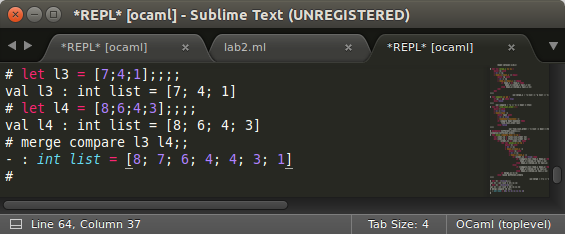
Тест 2: “тестирование обобщенной функции **merge** с двумя списками отсортированных в порядке невозростания (оба)”



Как видно, здесь выполняется условие (order\_lx == false), которое позволяет получить отсортированный список в порядке невозрастания.

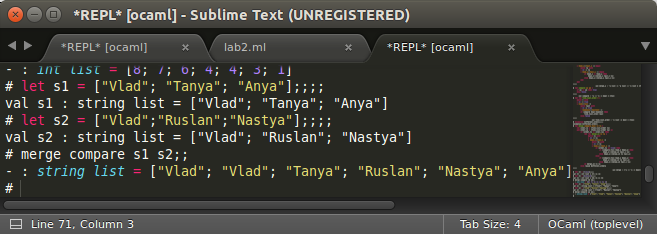
## Задача 3

Тест 1: тестирование обобщенной функции merge на двух списках целых чисел отсортированных в порядка невозростания



В этом тесте сортировка выполняется правильно, благодаря функции find\_list\_order, которая определяет порядок сортировки списков.

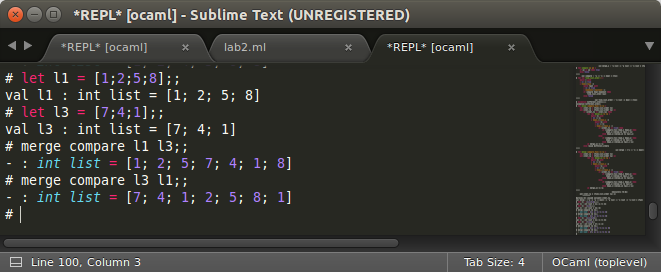
Тест 2: “тестирование обобщенной функции merge на двух списках строк отсортированных в порядке невозростания”



Как видим, функция compare поддерживает даже лексикографический порядок - сравнивает.

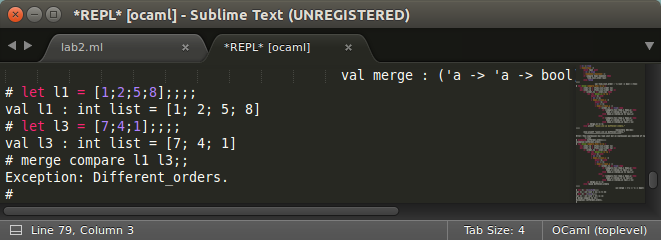
## Задача 4

Тест 1: “тестирование функции merge на списка отсортированных в разных порядках”



В данном тесте мы рассматриваем два примера. Обычная обобщенная функция merge комбинирует два списка учитывая только порядок первого списка

В итоге я решил создать исключение, которое будет генерироваться при попытке комбинирования двух списков с разными порядками сортировок:

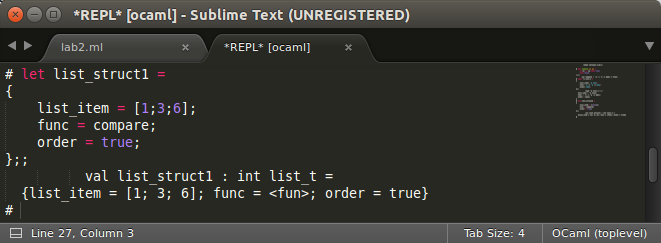


Такой способ позволяет избежать неправильного комбинирования списков с разными порядками.

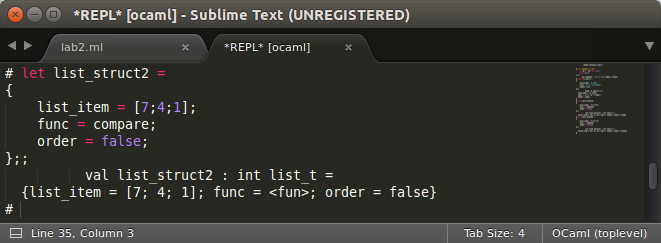
## Задача 5

Тест: “создание записей типа list\_t”

В неубывающем порядке:

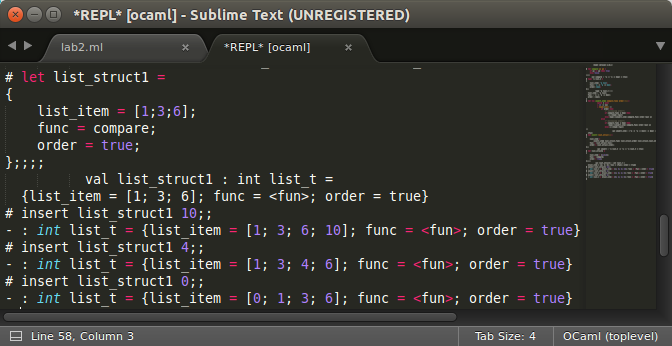


В невозрастающем порядке:



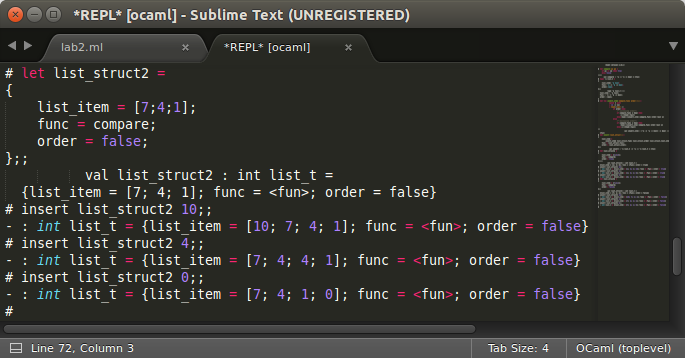
## Задача 6

Тест 1: “тестирование функции insert на списке (list\_item) с неубывающим порядком сортировки”



В данном случае, мы пытаемся добавить в массив [1;3;6] по-очереди элементы 10, 4 и 0. В каждом из случаев элементы добавляются в массив не нарушая исходный порядок сортировки указанный в order (order=true в данном случае) нашей записи. При самом добавлении создаётся новый массив (как и новая запись), количество элементов которого ([1;3;6;10]) больше предыдущего ([1;3;6]).

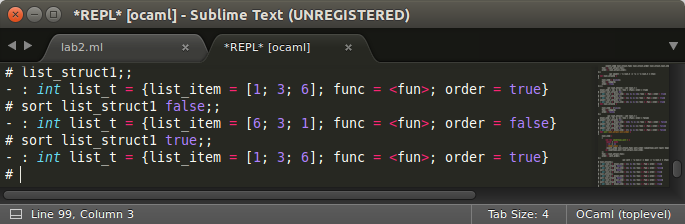
Тест 2: “тестирование функции insert на списке (list\_item) с невозрастающим порядком сортировки”



На примере показано, что и при невозрастающим порядке сортировки новые элементы добавляются в массив list\_item не нарушая этот порядок.

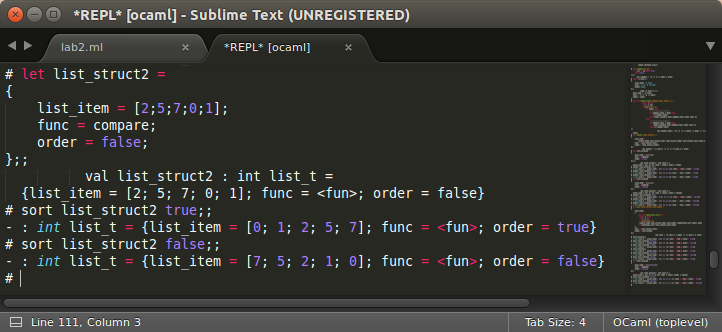
## Задача 7

Тест 1: “тестирование функции sort для списка (insert\_item) отсортированного в неубывающем / невозрастающем порядке”



При указании порядка сортировки через false, список list\_item в list\_struct1 сортируется в невозрастающем порядке, а при указании true - в неубывающем.

Тест 2: “тестирование функции sort на не отсортированном списке”



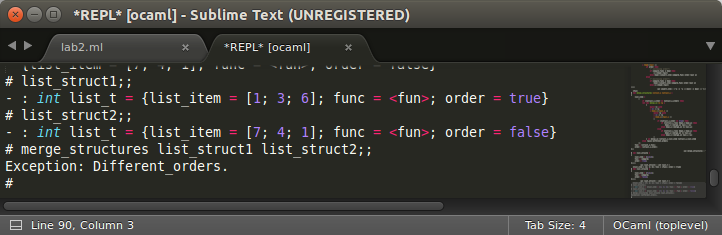
Здесь мы имеем запись с массивом [2;5;7;0;1] и со значением параметра order - false. false якобы означает, что массив отсортирован в невозрастающем порядке, но это не так. В данном случае не важно какое значение принимает order. Важно то, что при использовании функции sort мы непосредственно указываем требуемый порядок сортировки:

* true - если нужен неубывающий порядок (как после первого использования sort);
* false - если невозрастающий порядок (как после второго использования sort).

После выполнения sort будет создана новая запись и уже с новым значением параметра order, которое будет соответствовать введённому значению (true/false) при вызове sort.

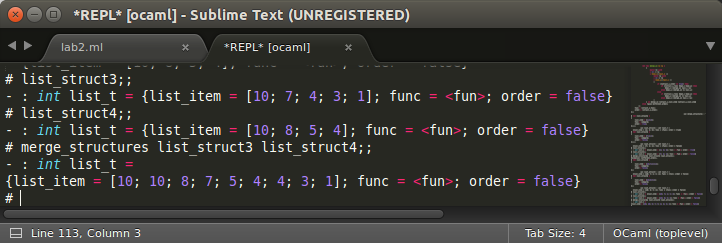
## Задача 8

Тест 1: “тестирование merge\_structures на двух записях с разными значениями order (разными порядками сортировок списков list\_item)”



При попытки комбинировать два списка с разными порядками сортировок интепретатор выдаст то самое исключение, которое упоминалось в предыдущих пунктах.

Тест 2: “тестирование merge\_structures на двух записях с одинаковыми значениями order”



Как видно, на двух записях с одинаковыми значениями order функция merge\_structures работает корректно: порядок сортировки не нарушается, а list\_item в новой записи содержит суммарное количество элементов, учитывая два предыдущих списка.

# 

# 

# Анализ ошибок (описание устранения замечаний)