Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

Факультет «Информатика и системы управления»

Кафедра «Системы обработки информации и управления»

ОТЧЁТ ПО

Домашнему заданию

Выполнил: Шибанова Варвара

студент группы ИУ5-31Б

_____

Подпись и дата:

Проверил:

преподаватель каф. ИУ5

Нардид А.Н.

Подпись и дата:

Москва

2024 г

Оглавление

Введение

Язык программирования OCaml (Objective Caml) занимает уникальную нишу в мире разработки программного обеспечения. В отличие от широко распространенных языков общего назначения, таких как Python или Java, OCaml фокусируется на обеспечении высокой надежности, эффективности и безопасности кода, что делает его идеальным выбором для проектов, где ошибки недопустимы. В данной работе будут рассмотрены особенности OCaml, его сильные стороны и области применения.

OCaml относится к семейству языков ML (Meta Language), отличающихся строгими статическими типами и функциональным парадигмой программирования. Строгая типизация позволяет компилятору выявлять множество ошибок еще на этапе написания кода, предотвращая сбои и непредсказуемое поведение программы во время выполнения. Это существенно повышает надежность и безопасность разрабатываемого программного обеспечения, что особенно важно в критически важных системах. Функциональный подход, основанный на использовании неизменяемых данных и чистых функций, упрощает разработку сложных алгоритмов, делает код более предсказуемым и облегчает его тестирование и отладку.

В отличие от императивных языков, где программа описывается как последовательность команд, изменяющих состояние данных, функциональный подход OCaml фокусируется на преобразовании данных с помощью функций. Это приводит к более модульному, читаемому и легко поддерживаемому коду. Функции в OCaml являются "гражданами первого класса", что означает, что они могут передаваться как аргументы другим функциям и возвращаться из них в качестве результатов. Это позволяет создавать гибкие и абстрактные конструкции, упрощая разработку и расширение программного обеспечения.

Функциональное программирование

1. Примитивные типы и величины

OCaml обладает строгими статическими типами. Основные типы данных включают:

В качестве основных типов используются:

Название	Обозначение	Пример
Целый	int	8
Вещественный	float	3.1
Логический	bool	true
Строковый	string	"Hello"
Символьный	char	'a'

Операции над целыми значениями:

Название	Обозначение	Пример
Сложение	+	8+6
Вычитание	-	3-4
Умножение	*	9*9
Деление	/	5/2
Остаток от деления	mod	5 mod 2

Операции над вещественными значениями:

Название	Обозначение	Пример
Сложение	+.	8 +. 6
Вычитание	-.	3 -. 4
Умножение	*.	9 *. 9
Деление	/.	5 /. 2
Возведение в степень	**	5 ** 2

Для работы с вещественными числами можно использовать встроенные функции:

Название	Обозначение	Пример
Ближайшее целое, превосходящее x	ceil	ceil x
Ближайшее целое, не превосходящее x	floor	floor x
Квадратный корень	sqrt	sqrt x
Экспонента	exp	exp x
Натуральный логарифм	log	log x

Название	Обозначение	Пример
Десятичный логарифм	log10	log10 x
Синус	sin	sin x
Косинус	cos	cos x
Тангенс	tan	tan x
Арксинус	asin	asin x
Арккосинус	acos	acos x
Арктангенс	atan	atan x

Для работы с логическим типом имеются логические операции:

Название	Обозначение	Пример
Отрицание	not	not true
Конъюнкция	&&	true && false
Дизъюнкция	ll	true ll false

Для сравнения значений между собой используют такие операции:

Название	Обозначение
Структурное равенство	=
Физическое равенство	==
Отрицание =	<>
Отрицание ==	!=

Объявление переменных: ***let*** служит для связывания значений с именами (идентификаторами). Эти имена затем можно использовать в дальнейшем коде для доступа к связанным значениям. Важно отметить, что в функциональном программировании, к которому относится OCaml, переменные обычно являются неизменяемыми (immutable) после их инициализации. Это означает, что после присвоения значения переменной, изменить это значение нельзя.

Пример:

```
# let x = 2.5;; → val x : float = 2.5
```

Здесь и далее стрелкой (→) будем заменять слова «будет выведено» или «получим» и так далее. Не будем также в дальнейшем писать знак приглашения *(#)*. Как видим, полученный результат означает: переменная *x* типа ***float*** имеет значение *2.5*.

При объявлении любых переменных и функций перед их именем должно присутствовать служебное слово ***let***. Тип ***string*** представляет набор символов в двойных кавычках, а тип ***char*** – единичные символы в одинарных кавычках.

2. Кортежи (tuple)

Кортежи представляют собой упорядоченные наборы значений различных типов. Они объявляются с помощью запятых, заключенных в круглые скобки.

Пример:

```
let a = (1,2,3,4);; →val a : int * int * int * int = (1, 2, 3, 4)
```

Иногда, если нет двусмысленности, скобки можно опускать.
Элементы могут быть разных типов:

Пример:

```
let x = 3, "Anna", true;; → val x : int * string * bool = (3, "Anna", true)
```

Обратите внимание, что в перечне типов элементов кортежа использован знак *(*)*. Особенно широко используются кортежи с двумя элементами, обычно называемые парами (*pairs*).

Пример:

```
let p = (5.078, "Hello");; → val p : float * string = (5.078, "Hello")
```

Для пар есть две встроенных функции: ***fst*** – возвращает первый элемент пары, и ***snd*** – соответственно, для второго элемента.

```
fst p;; → : float = 5.078
snd p;; → : string = "Hello"
```

Наконец, отметим, что элементами кортежей могут быть сами кортежи, а также списки, функции и вообще всё ,что угодно.

Пример:

```
let c = ("aaa", 2.3, (true, false));; → val c : string * float * (bool * bool) = ("aaa",
2.3, (true, false))
```

3. Списки

List – это последовательность элементов строго одного типа, разделённых знаком *(;)* и заключённых в квадратные скобки:

Пример:

```
let a = [1;2;3;4;5];; → val a : int list = [1; 2; 3; 4; 5]
```

Здесь указывается тип элементов и добавляется слово ***list***. Для доступа к элементам определены две функции: ***hd*** – возвращает первый элемент - голову (head) списка и ***tl*** – возвращает хвост (tail) – исходный список без первого элемента

Пример:

```
let h = List.hd a;; → val h : int = 1
let t = List.tl a;; → val t : int list = [2; 3; 4; 5]
```

Для пустого списка принято обозначение *[]*:

Пример:

```
let s = [];; → val s : 'a list = []
```

Инфиксный оператор *::* добавляет элемент в начало списка:

Пример:

```
let a = [1;2;3];;   let b = 82 :: a;; → val b : int list = [82; 1; 2; 3]
```

Поэтому можно создавать списки и таким образом:

```
let a = 2.1 :: 0.7 :: 12.4 :: [];; → val a : float list = [2.1; 0.7; 12.4]
```

Здесь мы последовательно добавляем элементы к пустому списку. Для конкатенации списков применяется знак *(@)*:

Пример:

```
let a = [1;2;3];; let b = [4;5];; let c = a @ b;; → [1; 2; 3; 4; 5]
```

Есть возможность создавать и вложенные списки:

```
let a = [1;2;3];;
```

```
let b = [4;5;6];;
let c = [a; b];; → val c : int list list = [[1; 2; 3]; [4; 5; 6]]
```

Для извлечения, например, первого элемента придётся поступить так:

```
List.hd(List.hd c);; →  1
```

Практически такие списки применяются не часто. Элементами списка также могут быть кортежи, функции и так далее.

4. Оператор if

Оператор ***if*** имеет привычный синтаксис:

```
if expr1 then expr2 else expr3
```

Выражение ***expr1*** должно возвращать результат типа ***bool***, а выражения ***expr2*** и ***expr3*** могут иметь любой, но только одинаковый тип.

Пример:

```
if 3=4 then o else 4 ;; → : int = 4
if 3=4 then "aa" else "bb" ;; → : string = "bb"
```

Результат, возвращаемый оператором *if* можно сразу использовать в вычислениях:

Пример:

```
let x = (if 3 = 5 then 2 else 7) + 9;; → val x : int = 16
```

5. Анонимные или абстрактные функции

Для создания таких функций используется служебное слово ***function*** и знак (->):

```
function x -> x * x ;; → : int -> int = <fun>
```

*(*Вместо имени функции служебное слово function, а вместо знака равенства — стрелка).*

Выражение после стрелки называют телом функции.

Пример:

```
let a = (function x -> x * x) 5;; → val a : int = 25
```

Все, что указывается после скобок будет рассматриваться, как аргумент функции. Аргументов может быть и больше одного, но тогда их надо объединять в кортеж.

Пример:

```
let a = (function (x, y) -> x * x + y) (2, 7);; → val a : int = 11
```

Всё аналогично и для именованной функции.

Пример:

```
let f x = (function y -> 3*x + y);; → val f : int -> int -> int = <fun> f 2 3;; →9
```

Следующие два объявления функции *f* фактически эквивалентны:

```
let f = function x -> function y -> 2*x + 3*y ;
let f x y = 2*x + 3*y ;
```

Количество аргументов у функции называют её арностью *(arity)*. У функции *f* арность равняется двум. Если мы используем кортеж, как уже делали это выше, то на самом деле получим функцию с арностью, равной единице.

Но в языке предусмотрена возможность создания анонимных функций с арностью больше единицы. Для этого надо использовать слово *fun* вместо *function:*

Пример:

```
fun x y -> 3*x + y ;; → : int -> int -> int = <fun>
```

тогда будет и соответствующий способ вызова:

```
(fun x y -> 3*x + y) 2 3 ;; → 9
```

Возвращаемые анонимными функциями значения в Ocaml называют *closure* (замыкание). Каждое выражение вычисляется в неком окружении *(environment)* содержащем ранее объявленные или вычисленные именованные значения.

Пример:

```
let m = 3 ;;
let f = function x -> x + m ;; → val f : int -> int = <fun> f 5;; → 8
```

closure нашей анонимной функции равно 8, а окружение составляет переменная *m*, равная 3.

Для переменных окружения различают статическую область видимости, ту, что была на момент объявления функции, и динамическую — на момент вызова функции.

6. Инфиксные функции и функции высшего порядка

Многие инфиксные функции, например такие, как арифметические операции, конкатенация, операции сравнения, логические операции и некоторые другие, могут быть представлены в обычной для всех функций префиксной форме. Для этого знак операции заключается в круглые скобки.

Пример:

```
( + ) 2 3;; → 5
( > ) 5 2;; → true
```

Можно использовать имя:

```
let f = ( + );; → val f : int -> int -> int = <fun> f 2 3;; → 5
```

Легко реализовать частичное применение:

```
let f = ( *. ) 3.5;; → val f : float -> float - f умножает аргумент на 3.5 f 2.0;; → 7.0
```

**(арность функции f теперь равна 1)*

Можно реализовать и обратное превращение функции к инфиксной форме, создавая новые функции такого типа.

Пример:

```
Let ( ++ ) c1 c2 = (fst c1)+(fst c2), (snd c1)+(snd c2) ;; → : int * int -> int * int -> int * int
let c = (2, 3) ;; - создали пару
c ++ c ;; →(4, 6) - сложили пару саму с собой
```

**(Используем знак (++) для операции поэлементного сложения двух кортежей).*

Рассмотрим более содержательный пример использования встроенной функции ***List.map*** для работы с элементами списка. Эта функция позволяет применять заданную функцию ко всем элементам списка и возвращает новый список.

Пример:

```
List.map;; → : ('a -> 'b) -> 'a list -> 'b list
```

Значит, эта функция принимает функцию, имеющую аргумент типа 'a и результат типа 'b, и список с элементами типа 'a, а возвращает список с элементами типа 'b.

Императивное программирование

В целях преодоления некоторых ограничений, присущих функциональным языкам, в Ocaml введены ряд императивных структур, позволяющих программирование в этом стиле.

1. Векторы

Векторы - те же списки, но с добавлением двух важных свойств: они имеют доступ к элементам по индексу и обладают изменяемостью *(mutable)*. Фактически это одномерные массивы *(array)* и это название будем применять наравне с «вектор». Для них принят такой, несколько необычный синтаксис:

Пример:

```
let v = [| 3.14; 6.28; 9.42 |] ;; → val v : float array = [|3.14; 6.28; 9.42|]
```

Объявить массив можно с помощью функции ***make*** из модуля ***Array***, принимающей два аргумента: число элементов массива (размер) и значение, которым эти элементы будут инициированы.

Пример:

```
let v = Array.make 3 3.14;; → val v : float array = [|3.14; 3.14; 3.14|]
```

Индексация элементов начинается с нуля, для извлечения *i-го* элемента принята форма — *v.(i)*:

```
let x = v.(0);; → val x : float = 3.14
```

Если указанный индекс выходит за пределы массива, генерируется исключение. Функция ***length*** возвращает размер массива.

Пример:

```
let n = Array.length v;; → val n : int = 3
```

Функция ***append*** применяется для объединения массивов.

Пример:

```
let a = Array.append v [|0.89; 0.33|];; → val a : float array = [|3.14; 3.14; 12345.75; 0.89; 0.33|]
```

Элементами вектора могут быть векторы и это позволяет создавать двумерные массивы.

Пример:

```
let m = [|1;2;3|];;
let n = Array.make 3 m;; → val n : int array array = [|[|1; 2; 3|]; [|1; 2; 3|]; [|1; 2; 3|]|]
```

Тем не менее, есть возможность работать с отдельными элементами таких двумерных массивов.

Функция ***copy*** – позволяет создавать копии.

Пример:

```
let b = Array.copy n;; → [|[|25; 2; 3|]; [|25; 2; 3|]; [|25; 2; 3|]|]
```

Переменные типа ***string*** обладают свойствами, во многом аналогичными свойствам векторов. В частности, для извлечения отдельного символа применяется похожий синтаксис.

Пример:

```
let s = "hello";; s.[2];; → : char = 'l'
```

Есть таже функция ***String.length***, определяющая количество знаков (размер) строковой переменной.

Пример:

```
String.length s;; → 5
```

2. Изменяемые поля

В OCaml по умолчанию поля записей (records) являются неизменяемыми (immutable). Это означает, что после создания записи её поля нельзя изменить. Это ключевой аспект функционального программирования, обеспечивающий предсказуемость и упрощающий рассуждения о поведении программы. Однако, в некоторых случаях может потребоваться изменять поля записей после их создания. Для этого в OCaml используется ключевое слово mutable.

Пример:

```
type point = { mutable x : float; mutable y : float };;
```

Теперь можно инициировать поля.

```
let p = { x = 1.0; y = 0.0 };; → val p : point = {x = 1.; y = 0.}
```

Для изменения полей тоже применяется обратная стрелка.

Пример:

```
p.x <- 3.0;; → : unit = ()[1]
p;; → {x = 3.; y = 0.}
```

Допустимо смешивать изменяемые и не изменяемые поля при определении записи, но модифицировать можно только изменяемые поля.

Пример:

```
type t = { c1 : int; mutable c2 : int } ;;
let r = { c1 = 0; c2 = 0 } ;; → val r : t = {c1 = 0; c2 = 0}
r.c1 <- 1 ;; → Ошибка, поле c1 неизменяемо
r.c2 <- 1 ;; → Всё в порядке
r ;; → {c1 = 0; c2 = 1}
```

3. Ввод – вывод

Для ввода-вывода в Ocaml есть две встроенные функции:

```
open_in;; → : string -> in_channel - для ввода
open_out;; → : string -> out_channel - для вывода
```

Обе эти функции принимают аргумент типа ***string*** и возвращают значение специального типа: ***in_channel*** — при вводе и ***out_channel*** — при выводе. Функция ***open_in*** открывает существующий файл для чтения или генерирует исключение, если такого файла нет.

Пример:

```
let ic = open_in "rab.txt";; → val ic : in_channel = <abstr>
```

Функция ***input_line*** читает одну строку файла и возвращает её в качестве переменной типа ***string***.

При попытке чтения после достижения конца файла генерируется исключение *«End_of_file»*.

Есть ещё одна функция ввода:

```
input;; → : in_channel -> bytes -> int -> int -> int
```

которая кроме переменной типа ***in_channel*** принимает ещё три аргумента: один типа ***bytes*** и два типа ***int***.

---

[1] Тип ***unit*** — это особый тип данных, который содержит только одно значение: (). Этот тип часто используется в ситуациях, когда функция не возвращает никакого осмысленного значения, но выполняет некоторое действие

Пример:

close_in ;; - **чтобы повторно читать из файла, его надо закрыть.**

let ic = open_in "rab.txt";; - **и снова открыть.**

let s :bytes = "kkkkkkkkkkkkkkkkk";; - **надо создать переменную типа bytes, заполнив её каким-то текстом.**

**(Размер этого текста должен быть не меньше того текста, который будет прочитан из файла)*.

Теперь можно вызвать функцию ***input***.

input ic s p l;;

Здесь p и l задают начальный и конечный номера символов, которые будут прочитаны из файла.

input ic s 0 5;; → : int = 5 **- возвращается количество прочитанных символов.**

Прочитанный текст содержится в переменной ***s***:

s;; → : bytes = "Hellokkkkkkkkkkk"

Функция ***input*** позволяет прочитать сразу весь текст файла, который будет представлен одной переменной типа ***bytes***.

Символы перевода на новую строку будут выведены в форме *"\n"*.

Вывод в файл выполняется аналогично.

Пример:

let oc = open_out "rab1.txt";; → val oc : out_channel = <abstr>

Будет создан и открыт для записи файл rab1.txt. Для вывода применяется функция ***output***:

output;; → : out_channel -> bytes -> int -> int -> unit

аргументы которой аналогичны аргументам функции ***input***.

В файл выводится содержимое переменной ***s*** типа ***bytes***, которая, значит, должна быть инициирована перед этим.

let s :bytes = "The most general

functions for reading";; **- текст может состоять из многих строк.**

output oc s 0 31;; → **В файл будут выведены знаки текста с номерами от 0 до 31. Функция вернёт количество переданных знаков.**

Как обычно, после вывода файл надо закрыть:

close_out;;

4. Циклы

Ocaml имеет две разновидности циклов в императивном стиле: цикл ***for*** и цикл ***while***.

for name = expr1 to expr2 do expr3 done;;

Слова ***for***, ***to***, ***do*** и **done** – ключевые. ***Name*** - идентификатор переменной цикла, которая имеет тип int и изменяется в цикле с шагом, равным единице. ***Expr1*** и ***expr2*** задают начальное и конечное значения для переменной цикла. Выражение ***expr3*** является телом цикла, оно также может быть представлено рассмотренной выше последовательностью. Выражение ***expr3*** должно иметь тип ***unit***, то-есть здесь может быть функция вывода или выражение, модифицирующее переменную. Слово ***done*** фиксирует конец тела цикла.

Пример:

*for i=2*3 to 5*2 do print_int i; print_string " " done;; → 6 7 8 9 10*

Цикл ***for*** является выражением и, в частности, сам может быть использован в последовательности.

Пример:

```
let x = for i=2*3 to 20 - 8 do print_int i; print_string " " done; 2.3;;
```

(Переменная x получает значение 2.3).

Ключевое слово ***to*** можно заменить на ***downto***, тогда переменная цикла будет изменяться в обратном порядке.

Пример:

```
for i=5 downto 1 do print_string (string_of_int i ^ ", ") done;; → 5, 4, 3, 2, 1,
```

Цикл ***while*** имеет такой синтаксис.

while expr1 do expr2 done

Здесь ***expr1*** — условное выражение (тип bool), тело цикла ***expr2*** будет повторно выполняться, пока выражение ***expr1*** возвращает значение ***true***. Также, как и в цикле ***for*** выражение ***expr2*** — тело цикла должно иметь тип ***unit***.

Пример:

```
let r = ref 1 in while !r < 11 do
print_string (string_of_int !r ^ ", "); r := !r + 1 done;; → 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

Отметим один интересный факт. Если объявить функцию, принимающую аргумент типа ***unit***, то этой функции можно передавать любое выражение типа ***unit*** и оно будет выполнено раньше, чем тело самой функции. Поскольку цикл ***for*** удовлетворяет этому требованию, его можно передавать, как аргумент.

Пример:

```
let f () = print_string "Hello world!\n";;
let s = for i=1 to 5 do print_string (string_of_int i ^ " ") done;; f s;; → 1 2 3 4 5 Hello world!
```

Объектно-ориентированное программирование

Язык программирования Ocaml обладает всеми необходимыми средствами для разработки программ в стиле ООП, что отражено и в самом названии языка - Objective Caml.

Классы

Для объявления класса принят в основном обычный синтаксис:

```
class name p1 p2. . . pn =
    object
        Тело класса
    end
```

Здесь ***name*** — имя класса (с маленькой буквы), ***p1 p2***. . . - параметры (аргументы) класса, которые, естественно, могут и отсутствовать. Ключевые слова ***object*** и ***end*** обозначают начало и конец тела класса (вместо фигурных скобок, принятых в большинстве других языков). В общем случае класс может содержать конструктор (явно не обозначается), поля и методы. Объявление поля начинается с ключевого слова ***val***.

```
val name = expr
```

Иногда поля объявляются изменяемые, для чего добавляется ключевое слово mutable.

```
val mutable name = expr
```

Соответственно, для объявления метода применяется ключевое слово method.

```
method name p1 p2. . . pn = expr
```

Параметры метода тоже могут отсутствовать.

Пример:

```
class point (xp, yp) =
    object
        val mutable x = xp
        val mutable y = yp
        method getx = x
        method gety = y
        method moveto (a,b) = x <- a ; y <- b
        method rmoveto (dx,dy) = x <- x +. dx ; y <- y +. dy
        method to_string () =
        "( " ^ (string_of_float x) ^ ", " ^ (string_of_float y) ^")" method dist =
    sqrt(x *. x +. y *. y)
    end;;
```

Библиотеки

В дистрибутиве Ocaml имеется довольно много библиотек, представленных в виде скомпилированных файлов

Для использования библиотеки принят обычный синтаксис с точкой.

Name.f – означает вызов объекта ***f*** из модуля ***Name***. Чтобы не повторять префикс при многократном вызове функций из одного и того же модуля Name, его можно открыть командой – *open Name;;* (После чего имя модуля можно не указывать).

Автоматически загружаемая библиотека ***Pervasives*** содержит объявления базовых типов, стандартные исключения и пару сотен функций. Большое количество модулей расположено в стандартной библиотеке. Кратко рассмотрим некоторые из них.

Модуль ***Random*** представляет генератор случайных чисел типов ***int*** и ***float***.

open Random;; **- эта команда ничего не возвращает**

Чтобы получить случайные числа, надо сначала выполнить инициацию.

Пример:

```
init 5;; → : unit = ()
full_init [|5; 2; 9|];; → : unit = ()
```

Аргументы у этих функций (в первом случае целое число, а во втором — вектор) задают определённую серию случайных чисел.

С некоторыми функциями модуля ***List*** мы уже знакомы: ***length***, ***hd***, ***tl***. Далее последуют примеры еще нескольких функций.

Функция ***nth*** обеспечивает доступ к любому элементу списка:

open List;; - (дальше будем всегда считать, что модуль уже открыт)

let a = [45; 2; 17; 34; 9];;

let x = nth a 3;; → val x : int = 34 - **(индексация начинается с 0)**

Функция ***rev*** (реверс) меняет порядок элементов в списке.

Пример:

rev a;; → [9; 34; 17; 2; 45]

Функция ***mem*** определяет принадлежность элемента списку.

Пример:

mem 17 a;; → true

Есть функции, осуществляющие поиск в списке заданных элементов. Так функция ***find*** возвращает первый же элемент в списке, удовлетворяющий заданному условию.

val find : ('a -> bool) -> 'a list -> 'a

Пример:

```
let a = [7.4; 2.3; 5.8; 2.3; 23.0; 2.3];;
find (fun x -> x>3.) a;; → 7.4
find (fun x -> x<3.) a;; → 2.3
```

Функция ***find_all*** возвращает список всех элементов, удовлетворяющих условию.

Пример:

```
find_all (fun x -> x>3.) a;; → [7.4; 5.8; 23.]
find_all (fun x -> x<3.) a;; → [2.3; 2.3; 2.3]
```

Функция ***partition*** возвращает пару из списков, удовлетворяющих и не удовлетворяющих условию.

Пример:

partition (fun x -> x>3.) a;; → ([7.4; 5.8; 23.], [2.3; 2.3; 2.3])

Имеется несколько функций, выполняющих такую важную операцию, как сортировка списков. Функция ***sort*** сортирует элементы списка по возрастанию.

Пример:

let a = [45.3; 12.7; 55.0; 1.02; 12.7; 0.5; 12.7];;
let b = sort compare a;; → [0.5; 1.02; 12.7; 12.7; 12.7; 45.3; 55.]

Здесь ***compare*** — стандартная функция, принимающая две переменных x и y и возвращающая -1 при x<y, 0 при x=y и 1 при x>y.

Для получения списка, отсортированного по убыванию, лучше всего воспользоваться функцией реверса ***rev***.

Пример:

let b = rev (sort compare a);; → [55.; 45.3; 12.7; 12.7; 12.7; 1.02; 0.5]

Функция ***sort_uniq*** удаляет дублирующиеся элементы в отсортированном списке.

Пример:

let b = sort_uniq compare a;; → [0.5; 1.02; 12.7; 45.3; 55.]

Функция ***merge*** принимает два сортированных списка и возвращает общий сортированный список.

Пример:

let b = [0.5; 1.02; 12.7; 12.7; 12.7; 45.3; 55.];;
let c = [1.3; 34.8; 66.35; 87.2];;
merge compare b c;; → [0.5; 1.02; 1.3; 12.7; 12.7; 12.7; 34.8; 45.3; 55.;66.35; 87.2]

В модуле ***String*** имеется много функций для обработки строк.

Примеры:

open String;;
let s = "Hello world";; length s;; → **11**
get s 4;; -> *'o'* - **вернула 4-й символ, нумерация с 0**
make 5 'd';; → "ddddd" - **возвращает строку**
init 7 (get s);; → "Hello w" - **принимает число и функцию get**
sub s 3 5;; → "lo wo" - **вернула 5 знаков, начиная с 3 позиции**
concat s ["{"; "}"];; → "{Hello world}" - **принимает строку и список**
iter print_char s;; → Hello world — **принимает функцию, что-то делающую над символами**

let s1 = "Hello world\n";;
trim s1;; → "Hello world" — **удаляет символ перевода строки "\n", а также "\t", "\r" и некоторые другие**

Заключение

В заключение следует отметить, что знакомство с языком программирования OCaml открывает перед разработчиком новые горизонты в создании надежного, эффективного и безопасного программного обеспечения. Несмотря на то, что OCaml не является самым популярным языком общего назначения, его строго типизированный функциональный подход и мощные средства метапрограммирования делают его незаменимым инструментом в ряде специфических областей.

Изучение OCaml, хоть и может показаться сложным на начальном этапе, но оно приносит значительную отдачу в виде глубокого понимания принципов функционального программирования и навыков создания высококачественного кода. Строгая система типов, хотя и требует большего внимания к деталям при написании кода, надежно предотвращает целый класс ошибок, типичных для динамически типизированных языков. Это приводит к более надежному и предсказуемому поведению программ, что критически важно в проектах, где сбои недопустимы.

В ходе изучения данного языка программирования и написания домашнего задания были рассмотрены основные синтаксические конструкции OCaml, его основные типы данных и способы их обработки. Были изучены ключевые особенности функционального программирования, а также небольшой список возможно используемых библиотек.

Литература

1. Chailloux E., Manoury P., Pagano B. Разработка программ с помощью Objective Caml, (русский перевод), 2007.
2. Ценности и функции · Документация OCaml [Электронный ресурс] – Режим доступа: https://ocaml.org/docs/values-and-functions