# Лабораторная работа №2
# «Uno»

## Предварительные сведения

В ходе выполнения данной работы необходимо выполнить 40 заданий на языке Standard ML. Большинство из описываемых функций связано с реализацией карточной игры «Uno», правила которой (с некоторыми изменениями относительно классической версии) изложены в разделе 1. Непосредственно задания приведены в разделе 2.

Прежде чем приступать к выполнению заданий, ознакомьтесь с замечаниями, изложенными в пункте 3 данного описания.

## 1. Правила игры «Uno»

### 1.1. Колода для игры «Uno»

Игра проводится с колодой карт из 108 листов. Все карты делятся на карты по мастям и дикие карты. Масти — цвета: красный (RED), зеленый (GREEN), синий (BLUE), желтый (YELLOW). Дикие карты — черного цвета (BLACK).

В колоде присутствуют карты:

**Цифровые карты.** 76 карт. В каждом цвете по две карты номинала от 1 до 9 и одна карта номиналом 0. Стоимость карты — по номиналу — от 0 до 9 очков.

**Активные карты.** 24 карты. В каждом цвете по две карты каждого из трех видов: «SKIP» (Пропусти ход), «DRAW TWO» (Возьми две), «REVERSE» (Наоборот). Стоимость каждой карты — 20 очков.

**Дикие карты.** 8 карт. По четыре карты каждого из двух видов: «WILD» (Закажи цвет) и «WILD DRAW FOUR» (Закажи цвет и возьми четыре). Стоимость каждой карты — 50 очков.

### 1.2. Правила раздачи карт

В начале игры каждому игроку раздаётся 7 карт. Остальные карты кладутся рубашкой вверх — это колода «Прикуп» (deck). Верхняя карта из колоды «Прикуп» переворачивается, кладётся рядом и становится первой картой колоды «Сброс» (pile).

Если верхней картой колоды «Сброс» выпадает дикая карта, то она возвращается в колоду «Прикуп», карты колоды «Прикуп» тасуются и снова верхняя карта становится первой картой колоды «Сброс» (процесс повторяется, пока первая карта колоды «Сброс» — дикая).

Если верхняя карта колоды «Сброс» не REVERSE, то игра начинается по часовой стрелке и первым ходит игрок после раздающего. Иначе — игра начинается против часовой стрелки и тот, кто раздает, делает первый ход.

Если первая карта колоды «Сброс» — SKIP или DRAW TWO, то соответствующее действие должен выполнить игрок по левую руку от раздающего.

### 1.3. Ход игры

Во время своего хода игрок должен выложить одну карту на колоду «Сброс» по одному из следующих правил:

- карта должна быть того же цвета, что и верхняя карта на колоде «Сброс»;
- карта должна иметь ту же цифру или ту же картинку (быть активной картой), что и верхняя карта на колоде «Сброс»;
- карта — Дикая карта.

При отсутствии подходящий карты игрок берёт одну карту из колоды «Прикуп». Если карта удовлетворяет указанным выше условиям — игрок должен выложить карту на колоду «Сброс», если не удовлетворяет — ход переходит к следующему игроку.

Игрок не должен брать карт из колоды «Прикуп», если у него на руках имеется карта, которую он может положить в колоду «Сброс».

Если игрок выкладывает на колоду «Сброс» активную карту или дикую карту, то ход игры меняется:

**SKIP** — следующий игрок (по ходу игры) должен выложить свою карту SKIP, не беря карт из колоды «Прикуп», или пропустить свой ход.

**DRAW TWO** — следующий игрок (по ходу игры) должен выложить свою карту DRAW TWO, не беря карт из колоды «Прикуп», или взять две карты из колоды «Прикуп» и пропустить свой ход.

**REVERSE** — направление хода игры меняется на противоположное. Например, если игра велась по часовой стрелке, то после выкладывания карты будет вестись против часовой стрелки.

**WILD** — игрок, выкладывающий эту карту, заказывает цвет, которым должна быть покрыта эта карта. Дальше, по ходу игры на эту карту нужно положить любую карту заданного цвета или другую дикую карту (заказывая, возможно, другой цвет).

**WILD DRAW FOUR** — игрок, выкладывающий эту карту, заказывает цвет, которым должна быть покрыта эта карта. При этом следующий игрок должен взять из колоды 4 карты и пропустить ход. Следующий за ним игрок должен положить любую карту заданного цвета или другую дикую карту (как после карты WILD).

Игрок не может пасовать, если у него на руках имеется карта, которую он может положить в колоду «Сброс».

Никто из игроков не может выкладывать карту WILD DRAW FOUR, если у него есть карта того цвета, который требуется для хода. За нарушение этого правила игрок, выложивший эту карту, берет 4 карты из колоды сброс, а уже выложенная WILD DRAW FOUR действует так же, как карта WILD.

### 1.4. Окончание игры

Игра продолжается до тех пор, пока кто-то один из игроков не скинет все карты. После этого происходит подсчёт очков по оставшимся на руках картам. Сумма очков каждого игрока — его проигрыш победителю. Победитель тот, кто сбросил все свои карты.

Если последней картой, выложенной победителем, является WILD DRAW FOUR, то перед подсчетом очков следующий игрок должен взять 4 карты.

## 2. Задания

В шаблоне к решению приведены некоторые вспомогательные функции, структуры данных для разрабатываемой игры «Uno» и определен набор функций (конструкторов, геттеров, сеттеров и функций сравнения) для элементарных операций с этими структурами данных. Прежде чем выполнять последующие задания, внимательно изучите предложенные функции и конструкции.

Вспомогательная функция `seed()` выдает некоторое целое число в зависимости от текущего системного времени. Такое число понадобится нам для генератора случайных чисел при смешивании карт в колоде.

Вспомогательная функция `curry_2` — каррированная функция, получающая некаррированную функцию двух аргументов и два ее аргумента. С помощью этой функции можно превратить некаррированную функцию двух аргументов в каррированную для дальнейшего частичного применения.

Типы данных `color` и `rank` определяют, соответственно, цвет и значение карт. Предполагается, что цвет `BLACK` используется только для диких карт.

Карта представляется парой (цвет, значение), описанной типом `card` .

Тип `move` определяет возможные состояния очередного хода:

`PROCEED` — игрок должен сделать обычный очередной ход;

`EXECUTE` — верхняя карта колоды «Сброс» — активная и нужно выполнить заданное ею действие;

`GIVE col` — верхняя карта колоды «Сброс» — дикая и ее нужно накрыть картой цвета `col` (или другой дикой).

Тип `strategy` задает сигнатуру функции, определяющей стратегию игрока. Считаем, что игрок знает состояние хода (значение типа `move`), видит свои карты (список типа `card list`), верхнюю карту колоды «Сброс» (значение типа `card`) и знает количество карт у других игроков (список типа `int list`). Предполагается, что функция запускается, если среди карт игрока есть карта, которой можно сделать ход. Функция должна выбирать карту для хода, а если выбирается дикая карта, то еще и задавать нужный цвет, т. е. возвращать пару типа `card * color`, в которой второй элемент имеет определенный смысл только если первый элемент — дикая карта).

Тип `player` описывает игрока и определяется в виде записи из трех полей: `name` — строка, задающая имя игрока, `cards` — список карт игрока и `strat` — функция стратегии игрока.

Тип `desk` — запись из четырех полей, задающая текущую конфигурацию игры: `players` — список игроков (в порядке хода игры), в котором первый элемент — игрок, чей ход очередной; `pile` — список карт колоды «Сброс»; `deck` — список карт колоды «Прикуп»; `state` — состояние текущего хода. Порядок карт в списках полей `pile` и `deck` — от верхней карты к нижней.

Исключение `IllegalGame` — для исключения невозможных ситуаций в игре, `IllegalMove name` — для исключения некорректных ходов игрока с именем `name`.

Функции `isSameRank`, `isSameColor`, `isSameCard` — функции сравнения на равенство двух значений карт, двух мастей, двух карт соответственно.

В списке `ranks` перечислены все возможные значения карт в колоде для игры.

Функция `makePlayer` создает элемент типа `player` из трех составляющих (конструктор). Функции `getPlayerName`, `getPlayerCards`, `getPlayerStrategy` — селекторы (геттеры) соответствующих полей записи типа `player`. Функция `setPlayerCards` выдает исходное значение типа `player` с заменой в нем списка карт игрока на заданный (сеттер).

Функция `makeDesk` является конструктором для элемента типа `desk`, с функциями `getDeskPlayers`, `getDeskPile`, `getDeskDeck`, `getDeskState` в качестве геттеров и `setDeskPlayers`, `setDeskState` в качестве сеттеров.

Функция `falseStrategy` представляет из себя «фальшивую» стратегию игрока — вспомогательную функцию для отладки и тестирования функций игры. Стратегия выбирает для очередного хода первую карту из списка карт игрока при любом раскладе игры, даже если такой ход не является корректным.

1. Опишите функцию `getNth`, получающую список и целое `n`. Функция должна вернуть `n`-ый элемент заданного списка, при условии, что элементы нумеруются с нуля. В случае, когда число `n` отрицательное или превышает количество элементов заданного списка, функция должна создавать исключение `List.Empty`.

   Типовое решение — 3–4 строки кода.

2. Опишите функцию `reverseAppend`, получающую два списка и возвращающую список, полученный в результате вставки элементов первого списка в обратном порядке во второй список. То есть, например, если аргументы — списки `[1,2,3]` и `[4,5,6]`, то результат — `[3,2,1,4,5,6]`.

   Решение — 2–3 строки.

3. Опишите функцию `cardValue`, принимающую карту в качестве аргумента и выдающую ее стоимость. Стоимость каждой карты описывается в разделе 1.1.

   Объем решения — 3–4 строки.

4. Опишите функцию `cardCount`, принимающую карту в качестве аргумента и выдающую общее количество экземпляров такой карты в колоде. О количестве экземпляров каждой карты говорится в разделе 1.1.

   Объем решения — 3–4 строки.

5. Опишите функцию `rankColors`, принимающую значение карты в качестве аргумента (элемент типа `rank`) и выдающую список возможных цветов (элементов типа `color`) карты с заданным значением. О том, какие карты могут принимать какие цвета, говорится в разделе 1.1.

   Объем решения — 3–4 строки.

6. Опишите функцию `sumCards`, получающую список карт в качестве аргумента и возвращающую сумму очков по данному списку. Для решения потребуется описать вспомогательную рекурсивную функцию, использующую аккумулятор (сумматор) в качестве дополнительного аргумента.

   Решение составит порядка семи строк.

7. Опишите функцию `removeNth`, получающую в качестве аргументов список и номер элемента и возвращающую данный список, в котором удален элемент с указанным номером. Считаем, что нумерация элементов в списке ведется с нуля. Если в качестве второго аргумента задан номер несуществующего в списке элемента, функция должна создавать исключение `List.Empty`.

   Для решения потребуется описать вспомогательную рекурсивную функцию со списком-аккумулятором и счетчиком в качестве дополнительных аргументов, использующую функцию `reverseAppend` для получения окончательного результата.

   Объем решения составит порядка восьми строк.

8. Опишите функцию `removeCard`, получающую в качестве аргументов список карт `cs`, карту `c` и исключение `e`. Функция должна вернуть данный список, в котором удалено первое вхождение карты `c`. Если карта `c` отсутствует в списке `cs`, функция должна создавать исключение `e`.

   Решение аналогично решению задания 7, но вместо проверки значения счетчика здесь очередная карта списка должна сравниваться с картой `c` с помощью функции `isSameCard`.

   Объем решения составит порядка девяти строк.

9. Опишите функцию `insertElem`, принимающую в качестве аргументов список `cs`, элемент `c` и номер `n`. Результатом функции должен быть список `cs`, в котором на позицию `n` вставлен элемент `c`. Если `n` — номер недостижимой позиции (больше чем длина списка `cs` или отрицательное число), то функция должна создавать исключение `List.Empty`.

   Решение аналогично решению задания 7. Объем решения составит порядка восьми строк.

10. Опишите функцию `interchange`, получающую в качестве аргументов список `cs` и номера двух элементов в нем — `i` и `j`. Функция должна возвращать список `cs`, в котором элементы с заданными номерами меняются местами.

    Для решения следует использовать ранее описанные функции `getNth`, `removeNth`, `insertElem`.

    Объем решения составит порядка 8 строк.

11. Опишите функцию перемешивания элементов списка `shuffleList`, получающую список в качестве аргумента и возвращающую список этих же элементов, но в случайном порядке.

    Для перемешивания элементов списка воспользуемся следующим алгоритмом: пусть `n` — количество элементов в заданном списке, тогда для каждого `i` от `0` до `n-2` сгенерируем случайное целое `j` в диапазоне от `i` до `n-1` и поменяем местами `i`-й и `j`-й элементы в списке.

    Для выполнения задания потребуется реализация вспомогательной процедуры, моделирующей цикл по `i`. Для перемены мест элементов списка следует использовать функцию `interchange`, реализованную ранее.

    Кроме этого, для генерации случайных чисел потребуются функции модуля `Random` — `Random.rand` и `Random.randRange` (порядок их использования см. на форуме на портале) и функция `seed` для генерации аргумента функции `Random.rand`.

    Объем форматированного решения — порядка 13 строк.

12. Опишите функцию `allRankColors`, получающую в качестве аргумента значение карты `r` (типа `rank`) и возвращающую список карт со значением `r` всех возможных для этого значения цветов (по одной карте каждого цвета).

    Для решения рекомендуется использовать ранее описанную функцию `rankColors`, к результату которой применить функцию `map` со вспомогательной неименованной функцией, превращающей цвет в карту со значением `r`.

    Типовое решение — 2 строки.

13. Опишите функцию `copyCardNTimes`, которой передается карта `c`. Функция должна возвращать список, состоящий из количества повторений карты `c`, задаваемого функцией `cardCount`.

    Для решения стоит описать вспомогательную функцию, организующую итерационный процесс для формирования списка из нужного числа одинаковых элементов.

Объем типового решения — 7 строк.

14. Создайте переменную `deck`, которой присвойте вычисленный список — колоду из 108 карт для игры в «Uno». В качестве заготовки для вычислений стоит использовать список `ranks`, заданный в заготовке к решению.

    Для вычислений сначала для каждого значения карты стоит получить список возможных карт с помощью `map` и `allRanksColors`. Каждую полученную карту стоит преумножить нужное количество раз с помощью `map` и `copyCardNTimes`. Промежуточные результаты стоит объединять в один список с помощью `foldr` и операции `@`, превращенной в функцию от двух аргументов.

    Решение составит порядка 2–3 строк.

15. Опишите функцию `getSameRank`, которой передаются значение карты `r` и список карт `cs`. Функция должна выдавать список всех карт из `cs` со значением `r`.

    Решение заключается в использовании функции `List.filter` с вспомогательной функцией.

    Решение — 2 строки.

16. Опишите функцию `getSameColor`, получающую цвет карты `col` и список карт `cs` в качестве аргументов. Функция должна выдавать список всех карт цвета `col` из списка `cs`.

    Решение аналогично решению предыдущей задачи.

    Решение — 2 строки.

17. Опишите функцию `hasRank`, получающую аргументами значение карты `r` и список карт `cs`. Функция должна выдавать `true`, если в списке `cs` есть карта со значением `r`.

    Решение заключается в использовании функций `isSome` и `List.find`, причем в `List.find` используйте для поиска ту же вспомогательную функцию, что и `List.filter` в задании 15.

    Решение — 2 строки.

18. Опишите функцию `hasColor`, получающую аргументами цвет карты `col` и список карт `cs`. Функция должна выдавать `true`, если в списке `cs` есть карта цвета `col`.

    Решение заключается в использовании функций `isSome` и `List.find`, причем в `List.find` используйте для поиска ту же вспомогательную функцию, что и `List.filter` в задании 16.

    Решение — 2 строки.

19. Опишите функцию `hasCard`, получающую карту `c` и список карт `cs` в качестве аргументов. Функция должна выдавать `true`, если в списке `cs` есть карта `c`.

    Решение практически повторяет решения предыдущих двух задач, но может быть несколько элегантней, если использовать функцию `curry_2` и `isSameCard` для получения предиката функции `List.find`.

    Решение — 2 строки.

20. Опишите функцию `countColor`, получающую аргументами цвет карты `col` и список карт `cs`. Функция должна выдавать количество карт цвета `col` в списке `cs`.

    Решение очень простое, если использовать функцию `length` и `getSameColor`.

    Решение — 1–2 строки.

21. Опишите функцию `maxColor`, получающую список карт `cs` и выдающую цвет (из всех цветов кроме черного), который встречается среди карт списка `cs` наибольшее количество раз. Если наибольшее количество раз в `cs` встречается несколько цветов, то функция должна выдавать тот из них, который в списке `[RED, GREEN, BLUE, YELLOW]` идет первым.

    В типовом решении три вспомогательные функции: первая формирует из списка `cs` список пар цвет-количество; вторая функция — функция сравнения двух пар цвет-количество, выдающая пару с максимальным количеством; третья функция, с помощью `foldl` и функции сравнения, находит в списке пар пару с максимальным количеством. Из результата последней функции извлекается необходимый цвет.

    Типовое решение — 13 строк.

22. Опишите функцию `deal`, раздающую карты игрокам. Функция должна получать в качестве аргумента список пар имя-стратегия (список типа `(string * strategy) list`), задающий порядок расположения игроков по часовой стрелке, начиная с раздающего. Результатом функции должна быть конфигурация игры сразу после раздачи (элемент типа `desk`). Функция должна сначала перетасовать целую колоду карт, после чего для каждого игрока сформировать элемент типа `player`, выделив ему 7 карт из колоды. Полученный набор элементов типа `player` составит список игроков. Из оставшейся колоды первая карта выделяется в `pile` (колоду «Сброс»), а все остальные карты образуют `deck` (колоду «Прикуп»). Из полученных элементов составляется элемент `desk`, в котором состоянием очередного хода устанавливается значение `PROCEED`.

    Для решения нужно описать вспомогательную функцию, организующую итерационный процесс обработки заданного списка. Можно использовать функции `List.take` и `List.drop` для выделения/удаления необходимого количества элементов списка.

    Задача данной функции — раздать игрокам карты. Элемент `desk`, возвращаемый функцией может оказаться некорректным, так как по правилам раздачи карт накладываются ограничения на карту в колоде `pile`. Проверку на корректность проводить здесь не следует: этим займется другая функция (задание 27).

    Типовое решение — 12 строк.

23. Опишите функцию `getPlayersFirst`, получающую элемент `dsk` типа `desk` и возвращающую текущего игрока (первый элемент списка `players` в записи `desk`).

    Типовое решение — 2 строки.

24. Опишите функцию `getPileTop`, получающую элемент `dsk` типа `desk` и возвращающую верхнюю карту колоды «Сброс» (первый элемент списка `pile` в записи `desk`).

    Типовое решение — 2 строки.

25. Опишите функцию `nextPlayer`, получающую и возвращающую элемент типа `desk`. Функция должна описать передачу хода от одного игрока к следующему: в списке `players` текущий игрок должен встать на последнюю позицию.

    При использовании в решении геттера `getDeskPlayers` и сеттера `setDeskPlayers` его объем составит порядка 4 строк.

26. Опишите функцию `changeDirection`, получающую и возвращающую элемент типа `desk`. Функция должна описать изменение порядка хода игры на противоположный (без смены текущего игрока): в списке `players` необходимо перевернуть хвост задом наперед.

    При использовании геттера `getDeskPlayers`, сеттера `setDeskPlayers` и функции `reverseAppend` объем решения составит 4 строки.

27. Опишите функцию `start`, получающую и возвращающую элемент типа `desk`. Функция должна корректировать состояние колод «Сброс» (поле `pile`) и «Прикуп» (поле `deck`), состояние очередного хода (поле `state`) и направление хода игры сразу после раздачи и передавать ход тому игроку, кто должен ходить первым. Таким образом, функция должна корректировать результат функции `deal` в соответствии с правилами раздачи карт, изложенными в разделе 1.2.

    Функция должна проверять верхнюю (и единственную) карту в списке поля `pile` и, если там дикая (черная) карта, то перетасовать ее вместе с колодой `deck` и выложить новую. Кроме того, если верхняя карта списка `pile` — карта `REVERSE`, функция должна поменять направление хода игры. В остальных случаях очередной ход должен быть передан следующему игроку, а в случае, если в `pile` активная карта (отличная от `REVERSE`), состояние очередного хода должно быть заменено на `EXECUTE`.

    Решение должно использовать функции `changeDirection`, `nextPlayer`, `shuffleList`, `setDeskState`.

    Объем решения — порядка 10 строк.

28. Опишите функцию `takeOne`, получающую и возвращающую элемент типа `desk`. Функция должна моделировать ситуацию, когда текущий игрок (первый игрок в списке поля `players`) берет из колоды «Прикуп» (список поля `deck`) одну карту. Если колода «Прикуп» пуста, то нужно взять все карты колоды «Сброс», кроме верхней карты, и переместить их в колоду «Прикуп», предварительно перетасовав.

Решение должно использовать функции `shuffleList` , `getPlayerCards` , `setPlayerCards` , `makeDesk` .

Объем типового решения — порядка 8 строк.

29. Опишите функцию `takeTwo` , получающую и возвращающую элемент типа `desk` . Функция должна моделировать ситуацию, когда текущий игрок берет из колоды «Прикуп» две карты.

    Решение должно использовать функцию `takeOne` и операцию композиции `o` .

    Объем типового решения — порядка 1 строка.

30. Опишите функцию `takeFour` , получающую и возвращающую элемент типа `desk` . Функция должна моделировать ситуацию, когда текущий игрок берет из колоды «Прикуп» четыре карты.

    Решение должно получаться из решения предыдущей задачи, если вместо функции `takeOne` использовать функцию `takeTwo` .

31. Опишите функцию `pass` , получающую и возвращающую элемент типа `desk` . Функция должна моделировать ситуацию, когда у очередного игрока нет карты, которой можно было бы сделать ход.

    Здесь может быть несколько ситуаций:

    - если наверху колоды «Сброс» карта `SKIP` , а состояние текущего хода — `EXECUTE` , то нужно передать ход следующему игроку и поменять состояние очередного хода на `PROCEED` ;
    - если наверху колоды «Сброс» карта `DRAW_TWO` , а состояние текущего хода — `EXECUTE` , то нужно выполнить действие, заданное картой, после чего передать ход следующему игроку и поменять состояние очередного хода на `PROCEED` ;
    - в остальных случаях нужно просто передать ход следующему игроку.

    Для решения стоит использовать функции `getDeskState` , `setDeskState` , `getPileTop` , `nextPlayer` , описанные ранее.

    Объем форматированного типового решения — 7 строк.

32. Опишите функцию `requiredColor` , получающую элемент типа `desk` в качестве аргумента. Функция должна выдавать цвет, который определяется текущей конфигурацией игры. Если состояние хода `GIVE col` , то `col` — результат функции. В противном случае результат функции — цвет карты наверху колоды «Сброс».

    Стоит использовать в решении функции `getPileTop` и `getDeskState` , описанные ранее.

    Типовое решение — 4 строки.

33. Опишите функцию `playableCards` , получающую элемент типа `desk` в качестве аргумента. Функция должна выдавать список тех карт, находящихся на руках у текущего игрока, которыми он мог бы сделать ход в данной конфигурации игры.

    Нужно рассмотреть несколько ситуаций:

    - состояние очередного хода — `GIVE col` : игрок может сделать ход только картой цвета `col` или дикой картой;
    - состояние очередного хода — `EXECUTE` : игрок может сделать ход только картой с тем же значением, что и карта наверху колоды «Сброс»;
    - состояние очередного хода — `PROCEED` : игрок может сделать ход дикой картой или картой с тем же значением или с тем же цветом, что и карта наверху колоды «Сброс».

    В типовом решении для каждой из ситуаций определяется предикат для фильтрования списка карт, который с помощью `List.filter` применяется к списку карт первого игрока.

    Для решения стоит использовать функции `isSameColor` , `isSameRank` , `getPlayersFirst` , `getPileTop` , `getDeskState` , `getPlayerCards` , описанные ранее.

    Объем форматированного типового решения — 15 строк.

34. Опишите функцию `countCards` , получающую элемент типа `desk` в качестве аргумента и выдающую список целых чисел — количество карт каждого игрока в порядке их следования по ходу игры.

    Решение состоит в том, чтобы извлечь список игроков с помощью `getDeskPlayers` , после чего к каждому элементу этого списка применить композицию функций `length` и `getPlayerCards` .

    Объем решения — 1–2 строки.

35. Опишите функцию `hasNoCards`, получающую элемент типа `player` в качестве аргумента и возвращающую `true`, если у игрока не осталось карт и `false` — в противном случае.

    Для решения стоит использовать функции `null`, `getPlayerCards` и операцию композиции `o`.

    Объем решения — 1 строка.

36. Опишите функцию `countLoss`, получающую список игроков (список типа `player list`) в качестве аргумента и подсчитывающую сумму очков по картам каждого игрока. Функция должна выдавать результат в форме списка пар имя-количество.

    Для решения стоит описать вспомогательную функцию, формирующую для одного игрока пару имя-количество, после чего с помощью `map` применить эту функцию к каждому элементу списка игроков.

    Потребуется использовать функции `getPlayerCards`, `getPlayerName`, `sumCards`, описанные ранее.

    Форматированное решение — 8 строк.

37. Опишите функцию для простейшей («наивной») стратегии игрока `naiveStrategy`. Как и любая стратегия, функция должна получать четверку элементов: состояние хода, список карт игрока, верхняя карта колоды «Сброс» и список, содержащий количество карт каждого игрока (включая текущего игрока). Функция должна выдавать пару карта-цвет, где карта — карта игрока, которой он делает ход, а цвет — цвет, который задается игроком в случае, если он делает ход дикой картой (в остальных случаях второй элемент пары не имеет значения).

    Предполагается, что функция будет запускаться только тогда, когда есть хотя бы одна карта, которой игрок может сделать ход.

    «Наивная» стратегия будет состоять в следующем:

    - В случае, когда состояние хода `GIVE col`:
      - если у игрока есть карты цвета `col` и карта `WILD`, а среди цветных карт (не черных) количество карт цвета `col` не является максимальным, то сделать ход картой `WILD` и заказать цвет максимального количества карт одного цвета;
      - если у игрока нет карт цвета `col` (значит, есть черные карты, т. к. существует хотя бы одна карта для хода), то пойти картой `WILD_DRAW_FOUR`, если она есть, а если нет, то картой `WILD`, при этом заказать цвет максимального количества карт одного цвета;
      - в остальных случаях (карты цвета `col` присутствуют у игрока) пойти любой картой цвета `col` с наибольшим количеством очков.
    - В случае, когда состояние хода `EXECUTE`: пойти любой картой с тем же значением, что и верхняя карта колоды «Сброс» (цвет значения не имеет).
    - В случае, когда состояние хода `PROCEED`:
      - если у игрока есть карты того же цвета или значения, что и верхняя карта колоды «Сброс», то выложить из них любую карту с наибольшей стоимостью.
      - если у игрока нет карт того же цвета или значения, что и верхняя карта колоды «Сброс» (значит, есть черные карты, т. к. существует хотя бы одна карта для хода), то пойти картой `WILD_DRAW_FOUR`, если она есть, а если нет, то картой `WILD`. При этом заказать цвет максимального количества карт одного цвета.

    Как видно, стратегия не использует информацию о количестве карт игроков. Основная ее идея — избавиться от карт с большим количеством очков.

    Для решения необходимо описать вспомогательные функции для поиска в списке карты с максимальным количеством очков (подобные функции уже должны быть описаны для поиска цвета максимального количества карт одного цвета). Для поиска цвета с максимальным количеством карт нужно использовать функцию `maxColor`. Кроме того, в решении стоит воспользоваться описанными ранее функциями `getSameColor`, `getSameRank`, `cardValue`, `hasRank`.

    Это максимальная по объему функция в данной лабораторной работе. Форматированное типовое решение — 37 строк.

38. Опишите функцию `play`, получающую в качестве аргументов элемент `dsk` типа `desk` и список `playCards` тех карт текущего игрока, которыми он может сделать ход (результат вызова `playableCards`).

    Функция должна моделировать ход текущего игрока. Предполагается, что `playCards` — непустой список.

    Результат функции — элемент типа `desk`, описывающий ситуацию в игре после хода игрока, или создание исключения, если стратегия игрока сделает недопустимый ход.

Необходимо извлечь функцию стратегии текущего игрока и запустить ее с необходимыми параметрами. Получив результат — карту и цвет, функция должна выполнить действие, соответствующее одному из следующих условий:

- если карта не является допустимой (отсутствует в списке `playCards`), то функция должна создавать исключение `IllegalMove name`, где `name` — имя игрока, сделавшего ход;

Во всех остальных случаях нужно удалить из карт игрока выбранную карту и поместить ее наверх колоды «Сброс». Кроме того,

- если карта — `WILD_DRAW_FOUR`, то следует установить состояние очередного хода `GIVE col`, где `col` — заказанный игроком цвет, и проверить, отсутствуют ли у игрока карты того цвета, который требовался для хода (правило, описанное в последнем абзаце раздела 1.3). Если такие карты найдутся, то следует отсчитать игроку 4 карты из колоды «Прикуп» и передать ход следующему игроку. В противном случае, отсчитать 4 карты очередному игроку и передать ход игроку, следующему за ним.
- если карта — `WILD`, то следует установить состояние очередного хода `GIVE col`, где `col` — заказанный игроком цвет, и передать ход следующему игроку.
- если карта цифровая, то следует установить состояние хода `PROCEED` и передать ход следующему игроку.
- если карта — карта `REVERSE`, то следует установить состояние очередного хода `PROCEED`, поменять направление хода игры на противоположное и передать ход следующему игроку.
- если карта активная, отличная от `REVERSE`, то следует установить состояние очередного хода `EXECUTE` и передать ход следующему игроку.

Для решения потребуются функции, описанные ранее: `getPlayerName`, `getPlayerStrategy`, `getPlayerCards`, `setPlayerCards`, `makeDesk`, `setDeskState`, `countCards`, `removeCard`, `hasColor`, `requiredColor`, `changeDirection`, `nextPlayer`, `takeFour`.

Форматированное типовое решение — 26 строк.

39. Опишите функцию `gameStep`, получающую и возвращающую элемент типа `desk`. Функция должна моделировать один ход в игре. Должен быть вычислен список карт, которыми текущий игрок может сделать ход. Если список не пустой, то нужно дать игроку сделать ход (с помощью функции `play`).

    В случае пустоты списка, если состояние хода `EXECUTE`, игрок должен пасовать (с помощью функции `pass`). Для других состояний — игрок должен взять одну карту из колоды «Прикуп» (с помощью функции `takeOne`), и если список карт, которыми можно сделать ход, останется пустым — пасовать, в противном случае — делать ход.

    Типовое решение — 18 строк.

40. Опишите функцию `game`, получающую элемент типа `desk` в качестве аргумента, как конфигурацию игры сразу после раздачи карт. Функция должна моделировать ход всей игры от момента раздачи карт до появления победителя — игрока, у которого не осталось карт. В результате функция должна выдать пару, в которой первый элемент — имя победителя, второй элемент — список пар имя-проигрыш.

    Сначала функция должна нормализовать начальную конфигурацию игры с помощью функции `start`. После чего должен запускаться итерационный процесс: выполнять функцию `gameStep` до тех пор, пока не появится игрок с пустым списком карт. Как только появится победитель, итерационный процесс должен остановиться и из заключительной конфигурации игры должен быть извлечен результат функции.

    Для реализации решения кроме уже перечисленных функций понадобятся `getDeskPlayers`, `hasNoCards`, `getPlayerName`, `countLoss` из описанных ранее и `List.find` из стандартных.

    Форматированное решение — 8 строк.

Общий объем решения, включая строки, предварительно заданные в файле, должен составить порядка 400 строк.

# 3. Замечания по выполнению заданий

## 3.1. Очередной минимум (в дополнение к перечисленному в лабораторной работе № 1)

Для выполнения работы потребуются сведения о следующих функциях, операциях и конструкциях:

- конструкция `case ... of ... => ... | ... => ...`
- конструкция описания исключения `exception` и функция создания исключения `raise`
- конструкции описания типов `type` и `datatype`

- функции для работы со случайными величинами `Random.rand` и `Random.randRange`
- функция для создания функционального значения `fn ... => ...`
- функции для агрегирования значений элементов списка `foldr`, `foldl`
- функция для обработки элементов списка `map`
- функции библиотеки `List`: фильтрации списка `List.filter`; поиска элемента `List.find`; извлечения подсписка из начала списка `List.take`; удаления подсписка в начале списка `List.drop`
- операция композиции функций `o`

Информацию об этих и других конструкциях можно найти в форуме на портале.

## 3.2. Ограничения

При выполнении данной лабораторной работы нужно соблюдать следующие ограничения:

- При описании функций НЕ ДОЛЖНО БЫТЬ ЯВНЫХ УКАЗАНИЙ ТИПОВ аргументов и результата (исключение — функции, приведенные в шаблоне решения);
- Нельзя использовать функции `#n`, `hd`, `tl`, `valOf`, `List.take`, `List.drop`, если использование этих функций не оговаривается специально в задании (вместо этого следует обходиться использованием шаблонов). Кроме того, нельзя определять и использовать аналоги этих функций;
- Нельзя использовать функции `getNth`, `removeNth`, `insertElem`, `interchange`, реализованные в заданиях 1, 7, 9 и 10, кроме тех случаев, когда их использование оговаривается в задании. Также нельзя использовать вспомогательные функции, подобные этим функциям;
- Операции сравнения на равенство `=` и неравенство `<>` можно использовать только для сравнения чисел (исключение — функции, приведенные в шаблоне решения);
- Нельзя использовать функции, не перечисленные в разделе 3.1 и подобном разделе предыдущей лабораторной работы. Если вы считаете, что для выполнения какого-то из заданий необходима особая функция или конструкция — задайте вопрос на форуме «Лабораторная работа №2»;

Можно определять любое количество собственных вспомогательные функции. Но вспомогательная функция может быть определена как функция верхнего уровня только если она используется в решениях минимум двух разных заданий. В остальных случаях вспомогательные функции должны быть локальными.

На код, реализующий тесты к функциям ограничений не накладывается.

## 3.3. Предостережения насчет решения

Решением каждой задачи должна быть функция с указанным именем и возвращающая значение в той форме, в которой спрашивается в задании. Прежде чем отправить решение на проверку проводите сравнение сигнатуры написанной вами функции с соответствующей сигнатурой, приведенной в разделе 3.4.

Не следует делать предположений насчет задания, не сформулированных явно в условии. Если возникают сомнения — задайте вопрос на форуме «Лабораторная работа №2».

Избегайте повторений вычислений. Вместо того, чтобы вычислять одно и то же значение несколько раз — сохраняйте вычисленное значение в переменной.

Избегайте повторений больших кусков кода. Вместо этого оформите такой кусок в виде вспомогательной функции, заменив одинаковые куски кода на ее вызов.

## 3.4. Результат

Запуск корректно выполненного решения должен привести к следующим определениям.

```
val getNth = fn : 'a list * int -> 'a
val reverseAppend = fn : 'a list * 'a list -> 'a list
val cardValue = fn : color * rank -> int
val cardCount = fn : color * rank -> int
val rankColors = fn : rank -> color list
val sumCards = fn : (color * rank) list -> int
val removeNth = fn : 'a list * int -> 'a list
val removeCard = fn : card list * card * exn -> card list
val insertElem = fn : 'a list * 'a * int -> 'a list
val interchange = fn : 'a list * int * int -> 'a list
val shuffleList = fn : 'a list -> 'a list
val allRankColors = fn : rank -> (color * rank) list
val copyCardNTimes = fn : color * rank -> (color * rank) list
val deck =
  [(Red,Num 0),(GREEN,Num 0),(BLUE,Num 0),(YELLOW,Num 0),(Red,Num 1),
   (Red,Num 1),(GREEN,Num 1),(GREEN,Num 1),(BLUE,Num 1),(BLUE,Num 1),
   (YELLOW,Num 1),(YELLOW,Num 1),...] : (color * rank) list
val getSameRank = fn : rank * ('a * rank) list -> ('a * rank) list
val getSameColor = fn : color * (color * 'a) list -> (color * 'a) list
val hasRank = fn : rank * ('a * rank) list -> bool
val hasColor = fn : color * (color * 'a) list -> bool
```

```
val hasCard = fn : card * card list -> bool
val countColor = fn : color * (color * 'a) list -> int
val maxColor = fn : (color * 'a) list -> color
val deal = fn : (string * strategy) list -> desk
val getPlayersFirst = fn : desk -> player
val getPileTop = fn : desk -> card
val nextPlayer = fn : desk -> desk
val changeDirection = fn : desk -> desk
val start = fn : desk -> desk
val takeOne = fn : desk -> desk
val takeTwo = fn : desk -> desk
val takeFour = fn : desk -> desk
val pass = fn : desk -> desk
val requiredColor = fn : desk -> color
val playableCards = fn : desk -> (color * rank) list
val countCards = fn : desk -> int list
val hasNoCards = fn : player -> bool
val countLoss = fn : player list -> (string * int) list
val naiveStrategy = fn : move * (color * rank) list * (color * rank) * 'a -> (color * rank) * color
val play = fn : {deck:card list, pile:card list, players:player list, state:move} * card list -> desk
val gameStep = fn : desk -> desk
val game = fn : desk -> string * (string * int) list
```

В зависимости от варианта решения ответ интерпретатора на определение некоторых функций может отличаться от вышеприведенных: вместо наименований типов `card`, `player`, `desk` может выдаваться их реализация (т. е., например, `color * rank` вместо `card`).