Лабораторная работа №3
«Uno»

1. Предварительные сведения

Лабораторная работа посвящена отработке следующих навыков

- написание программы в языке программирования с неявной типизацией;
- работа со сложными типами данных: списки, кортежи, структуры, контейнеры, пользовательские типы данных;
- использование функций высших порядков;
- сопоставление с образцом (шаблоном) при определении функций и инициализации имён;
- реализация итерационного процесса в виде функции с хвостовой рекурсией.

2. Общая схема разработки

В ходе настоящей работы мы реализуем движок для карточной игры UNO, правила которой (с некоторыми изменениями относительно классической версии) изложены в разделе 3. Решение всех заданий лабораторной работы позволит нам запускать и проводить игру как в автоматическом режиме, когда за игрока играет программа, так и в ручном режиме, когда действиями отдельных игроков управляет пользователь программы. Разрабатываемое приложение «расширяемое»: предусматривается возможность реализовать «особую» стратегию для каждого игрока в рамках предоставляемых параметров стратегии.

В ходе выполнения данной работы необходимо выполнить 32 задания на языке Standard ML. Непосредственно задания приведены в разделе 7.

Игра реализуется как процесс смены состояний или конфигураций игры — desk. Значение типа desk описывает ситуацию в игре в некоторый определённый момент времени. Оно содержит в себе список игроков, игровые колоды карт, значения состояния хода, направления хода игры и некоторый список последних ходов, являющийся видимой «памятью» для игроков.

Считаем, что в текущей конфигурации игры очередной ход должен делать тот игрок, которому в списке игроков соответствует первый элемент. Соответственно передача хода от одного игрока другому будет заключаться в изменении порядка элементов в списке игроков.

Каждый игрок, в свою очередь, имеет имя, список карт на руках и стратегию игры. С каждым ходом игры у игрока, делающего ход, меняется список карт.

Стратегия, которой руководствуется игрок в ходе партии определяется функцией стратегии `ownStrategy` какого-то модуля, удовлетворяющего сигнатуре `STRATEGY` (сигнатура определяет, что в модуле должна быть доступна только функция `ownStrategy`). Тривиальный пример такого модуля приводится среди предварительных определений (модуль False — фальшивая стратегия — неполноценная стратегия, которая по большей части не делает корректных ходов). Ещё один модуль со стратегией приведён в шаблоне решения — это модуль, функция стратегии которого реализует «ручное» управление картами игрока. Кроме того, одно из заданий посвящено разработке простенькой «наивной» стратегии игры для автоматической игры. В курсе планируется факультативное задание, в котором Вам будет предложено реализовать собственную функцию стратегии для игры в автоматическом режиме.

Запускает игру и реализует смену состояний игры функция `game`, которой передаётся подготовленный список игроков (каждому игроку предварительно даётся имя, стратегия и указывается будет ли игрок управляться вручную, вместо применения функции стратегии). Функция `game` раздаёт игрокам карты (запуская функцию `deal`) из перемешанной полной колоды карт `deck` формируя первоначальную конфигурацию игры, которую необходимо «нормализовать» (запуская функцию `deskNormalize`). После этого запускается цикл: оценивается возможность хода очередного игрока (функция `yourTurn`, а в ней для оценки понадобится функция `cardsToPlay`) и организуется смена состояния путём реализации хода игрока (функция `play`) или пропуска его хода с попутным «наказанием» (функция `execution`).

Во время хода игрока нам понадобится обращаться к его стратегии посредством функции `askPlayerForCard`, а затем реализовывать реакцию на ход с помощью функции `moveAction`.

Для полной реализации понадобится реализовать ещё пару десятков вспомогательных функций и воспользоваться теми вспомогательными функциями, что уже реализованы и предложены в дополнительном файле `lab-3-use.sml` (описание каждого определения в файле `lab-3-use.sml` приводится в комментариях в самом файле и в разделе 4).

Решения абсолютно всех заданий лабораторной работы являются частью реализуемой игры. Вся разработка ведётся в следующем порядке: сначала вспомогательные функции, а затем те, что их используют.

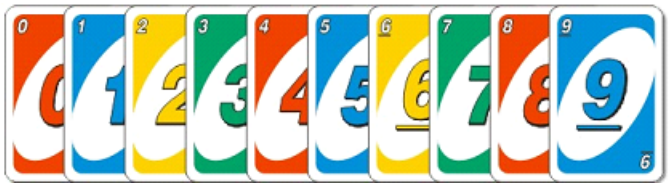
3. Правила игры «Uno»

3.1. 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`), жёлтый (`YELLOW`). Дикие карты цвета не имеют.

В колоде присутствуют карты:

Цифровые карты. 76 карт. В каждом цвете по две карты номинала от 1 (`NUM 1`) до 9 (`NUM 9`) и одна карта номиналом 0 (`NUM 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` » (Реверс). 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.

3.2. 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 раздаёт, делает первый ход.

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

3.3. Ход игры

Во время своего регулярного (ординарного) 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».

Игрокам не рекомендуется 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 «Прикуп», а уже выложенная `WILD_DRAW_FOUR` действует так же, как карта `WILD`.

3.4. 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 подсчётом очков следующий игрок должен взять 4 карты.

4. Предварительные определения

В файле `lab-3-use.sml` приведены определения функций и структур данных, необходимых для реализации лабораторной работы.

Начальный блок определений посвящен функциям для работы со списками. Первые две функции (`seed` и `interchange`) — вспомогательные для реализации функции `shuffleList`. Функция `shuffleList` принимает на вход произвольный список, а на выходе выдаёт список, в котором элементы исходного расположены в случайном порядке. Эта функция понадобится нам для «перетасовывания» колоды карт.

Функция `addNCopies : 'a * 'a list * int -> 'a list` возвращает список, полученный из исходного (из второго аргумента) добавлением в его начало n копий первого аргумента, где n — третий аргумент.

Функция `pass : 'a list * 'a list * int -> 'a list * 'a list` перекладывает поочерёдно n элементов из головы первого списка во второй список и выдаёт в результате пару получившихся списков. n — третий аргумент функции. Эта функция нам будет полезна при реализации раздачи карт игрокам.

Функция `shift : 'a * 'a list * int -> 'a list` вспомогательная для реализации записи в «память», имеющуюся в состоянии игры.

Следующий блок определений основной. Здесь определяются основные типы данных и элементарные операции над значениями этих типов.

Тип данных `color` определяет значения «масти» или цвета ординарной (не дикой) карты. Здесь же определяется список всех мастей `colors`, задающий среди мастей порядок «по возрастанию».

Тип данных `rank` определяет ранги или значения ординарной (не дикой) карты. Здесь же определяется список всех значений `ranks`, задающий среди значений карты порядок «по возрастанию».

Тип данных `card` с описывает возможные варианты карт. Имеется три варианта конструктора значения типа `card`: `WILD` и `WILD_DRAW_FOUR` для диких карт и `CRD : rank * color -> card`, получающий значение и цвет ординарной карты.

Тип данных `direction` описывает направления хода игры. В этом типе два значения: по часовой стрелке `CLOCKWISE` и против часовой стрелки `COUNTERCLOCKWISE`.

Тип данных `state` определяет возможные состояния очередного хода:

PROCEED — игрок должен сделать обычный очередной ход;

EXECUTE — верхняя карта колоды «Сброс» — активная (`SKIP` или `DRAW_TWO`) и нужно выполнить заданное ею действие;

контейнер `GIVE : color -> state` — верхняя карта колоды «Сброс» — дикая и ее нужно накрыть картой заданного цвета (или другой дикой).

Тип данных `move` определяет возможные варианты хода игрока:

контейнер `SIMPLE : card -> move` — игрок сделал обычный ход заданной картой;

контейнер `ORDER : card * color -> move` — игрок пошёл дикой картой (первый аргумент конструктора) и заказал цвет (второй аргумент конструктора);

`PASS` — игрок пропустил ход по тем или иным причинам (нечем ходить, взял карты из колоды «Прикуп»).

Далее в файле определяется тип `strategy` — синоним типа для функции стратегии.

Сигнатура `PLAYER` определяет интерфейс модуля `Player`. Модуль `Player` определяет функции для работы с игроком. Тип `player` определяется как запись, содержащая 4 поля: имя игрока, список карт игрока, функция стратегии игрока и отметка о том, управляется ли игрок вручную или для него сразу введён автоматический режим с использованием его функции стратегии. Конструктор `make` позволяет создать игрока из его составляющих. Набор селекторов позволяет для игрока узнать значения его составляющих. Кроме того задаются два сеттера: для смены набора карт игрока и для перевода игрока в автоматический режим.

Сигнатура `DESK` определяет интерфейс модуля `Desk`. Модуль `Desk` определяет функции для работы с конфигурацией игры (состоянием игры). Тип `desk` определяется как запись, содержащая 6 полей: список игроков, список карт колоды «Сброс», список карт колоды «Прикуп», состояние (тип) текущего хода, направление хода игры, список 10 последних ходов игроков. Конструктор `make` позволяет создать конфигурации игры из составляющих. Набор селекторов позволяет узнать значения составляющих конфигурации игры, а набор сеттеров позволяет изменить любое из полей. Дополнительно задаётся функция `whoseTurn : desk -> desk` которая не вносит изменений в переданную ей конфигурацию игры (то, что она получила на входе, она передаёт на выход), а только лишь выводит на экран сообщение о том игроке, чей очередной ход.

Исключение `IllegalGame` определено для поднятия в ситуациях, когда в игре возникает невозможная комбинация (например к началу очередного хода наверху колоды «Сброс» лежит карта `CRD (NUM 5, RED)`, а состояние хода — `GIVE BLUE`, состояние возможное только когда на колоде «Сброс» лежит дикая карта).

Исключение `IllegalMove` определено для поднятия в случае, если игрок делает некорректный ход, т. е., если функция стратегии игрока выдаёт некорректный результат (например, игрок делает ход `(SIMPLE (CRD (NUM 1, RED))` когда наверху колоды «Сброс» лежит карта `CRD (NUM 9, BLUE)`). Когда поднимается это исключение, ему передаются в качестве аргументов весь кортеж аргументов, который подаётся в функцию стратегии.

Сигнатура `STRATEGY` описана для определения и ограничения интерфейса модуля с функцией стратегии. Интерфейс такого модуля должен содержать только функцию `ownStrategy` типа `strategy`.

Примером модуля со стратегией может являться модуль `False` в котором определяется функция стратегии для тестирования функций, в которых стратегия может понадобиться (см. файл с примерами тестов). Функция `False.ownStrategy` предполагает, что первая карта в списке карт игрока — не дикая. Функция делает обычный ход картой с тем же рангом, что и первая карта, но зелёного цвета. Функция делает ход названной картой не зависимо от того, имеется ли эта карта на руках у игрока или отсутствует.

В лабораторной работе запрещено сравнивать на равенство нечисловые значения с помощью `=`, поэтому заранее определены и предлагаются для использования три типизированные функции сравнения на равенство значений карты (`isSameRank`), мастей (`isSameColor`) и карт (`isSameCard`).

Последний блок функций предназначен для перевода в строку значений и списков из значений, относящихся к определенным в этом файле типам данных.

5. Запуск игры

После реализации всех функций лабораторной работы можно запустить игру. Для этого нужно сначала сформировать список игроков.

Создать игрока можно с помощью функции `Player.make`, задав его имя, функцию стратегии и указав, будет ли игрок первоначально управляться вручную (из ручного управления всегда можно переключиться в автоматическое по ходу игры). Список карт игрока можно оставить пустым, так как он будет заменён при раздаче карт на актуальный.

Например,

```
val p1 = Player.make ("Sergey", [], Naive.ownStrategy, true)
```

Чтобы запустить игру, нужно функции `game` передать список готовых игроков. Например,

```
val res = game [p1, p2, p3, p4, p5, p6, p7, p8, p9, p10]
```

В файле `lab-3-game.sml` приведён пример инициализации и запуска игры.

При ручном управлении система ведёт диалог с пользователем, предлагая вести игру, выбирая карты из предложенных. Например

```
The turn of Sergey
State: GIVE RED
All cards on hand:
[ WILD_DRAW_FOUR
, CRD (REVERSE, BLUE)
, CRD (NUM 7, BLUE)
, CRD (NUM 9, YELLOW)
, CRD (NUM 4, YELLOW)
, CRD (NUM 3, YELLOW)
, CRD (NUM 2, YELLOW) ]
Playable cards:
[ WILD_DRAW_FOUR ]
A card on the pile: WILD
Game direction: CLOCKWISE
A list of last moves:
[ ORDER (WILD, RED)
, ORDER (WILD, RED)
, ORDER (WILD_DRAW_FOUR, RED)
, PASS
, SIMPLE (CRD (SKIP, BLUE))
, ORDER (WILD_DRAW_FOUR, BLUE)
, SIMPLE (CRD (NUM 4, RED))
, PASS
, SIMPLE (CRD (SKIP, RED)) ]
A list of the number of cards in the hands of the players:
[7, 6, 7, 6, 10, 6, 7, 10, 6, 6]
Enter the number of the selected card from the list of playable cards (from 0).
Any other number to switch to automatic mode.
```

Здесь на экран выводится имя игрока, делающего ход, и перечисляются все параметры функции стратегии: состояние игры (здесь `GIVE RED`), список всех карт на руках игрока, список карт, которыми игрок может пойти в этот ход (здесь в списке только одна карта `WILD_DRAW_FOUR`), карта, лежащая наверху колоды «Сброс» (здесь `WILD`), направление хода игры (здесь `CLOCKWISE`), «память» о 10 последних ходах и список количества карт на руках у игроков.

Выбор из списка карт осуществляется вводом номера выбранного элемента списка. Предполагается нумерация элементов списка с нуля. Таким образом, в приведённом примере нужно ввести 0 для выбора хода картой `WILD_DRAW_FOUR` и продолжения управления игроком в ручном режиме.

Выбор дикой карты приводит к необходимости выбора цвета для «заказа». Это производится аналогичным выбором элемента из списка. Например, если выбрать зелёный цвет (ввести 2), то может произойти следующий ответ системы.

```
Enter the number of color from list (from 0)
[YELLOW, BLUE, GREEN, RED]
Any other number to switch to automatic mode.
2
Sergey made move
ORDER (WILD_DRAW_FOUR, GREEN)

The turn of Pavel

The turn of Saveliy
Saveliy made move
SIMPLE (CRD (NUM 7, GREEN))

The turn of Igor
Igor made move
SIMPLE (CRD (NUM 8, GREEN))

The turn of Gennadiy
Gennadiy made move
SIMPLE (CRD (REVERSE, GREEN))

The turn of Igor
Igor made move
SIMPLE (CRD (NUM 7, GREEN))
```

```
The turn of Saveliy
Saveliy made move
SIMPLE (CRD (NUM 7, RED))

The turn of Pavel
Pavel made move
SIMPLE (CRD (NUM 6, RED))
```

Приведённый диалог говорит о том, что игрок Pavel пропустил ход (и получил на руки 4 карты), а последующие игроки сбросили по одной карте. Следующий запрос для игрока Sergey будет выглядеть так:

```
The turn of Sergey
State: PROCEED
All cards on hand:
[ CRD (NUM 1, RED)
, CRD (REVERSE, BLUE)
, CRD (NUM 7, BLUE)
, CRD (NUM 9, YELLOW)
, CRD (NUM 4, YELLOW)
, CRD (NUM 3, YELLOW)
, CRD (NUM 2, YELLOW) ]
Playable cards:
[ CRD (NUM 1, RED) ]
A card on the pile: CRD (NUM 6, RED)
Game direction: COUNTERCLOCKWISE
A list of last moves:
[ SIMPLE (CRD (NUM 6, RED))
, SIMPLE (CRD (NUM 7, RED))
, SIMPLE (CRD (NUM 7, GREEN))
, SIMPLE (CRD (REVERSE, GREEN))
, SIMPLE (CRD (NUM 8, GREEN))
, SIMPLE (CRD (NUM 7, GREEN))
, PASS
, ORDER (WILD_DRAW_FOUR, GREEN)
, ORDER (WILD, RED)
, ORDER (WILD, RED) ]
A list of the number of cards in the hands of the players:
[7, 9, 5, 4, 9, 6, 7, 10, 6, 6]
Enter the number of the selected card from the list of playable cards (from 0).
Any other number to switch to automatic mode.
```

Обратите внимание, что в предыдущий раз у игрока Sergey было 7 карт на руках, и в этот раз — опять 7 карт. Это означает, что перед текущим ходом Sergey взял одну карту из колоды «Прикуп» (та карта, которой он может пойти сейчас, отсутствовала у него в прошлый раз). Здесь в «памяти» можно обнаружить все те ходы, которые сделали игроки перед очередным ходом игрока Sergey после его последнего хода `ORDER (WILD_DRAW_FOUR, GREEN)`.

После окончания игры система выдаст результат (для вышеприведённого запуска игры)

```
val res =
  ("Arseniy",
   [("Sevastian",28),("Arseniy",0),("Olga",25),("Sergey",38),("Pavel",10),
    ("Saveliy",15),("Igor",28),("Gennadiy",90),("Irina",20),("Elsa",16)])
```

Результат означает, что победитель — Arseniy. Числа в паре с именами игроков — проигрыш соответствующего игрока победителю.

6. Замечания по выполнению заданий

6.1. Необходимый минимум

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

- конструкции `fun` и `val` для определения функций и переменных
- конструкция `if...then...else...`
- конструкция `let...in...end`
- стандартные арифметические и логические операции, стандартные операции сравнения
- конструктор кортежа `( , )`
- конструкторы списка `::` и `[]`
- операции для работы со списками: `null`, `@`
- функция преобразования целого числа в строку `Int.toString`

- конструкторы значений типа `option` — `SOME` и `NONE`
- функции для работы со значениями типа `option`: `valOf` и `isSome`
- конструкция `use` для загрузки функций из файла с заданным именем
- конструкции `structure` и `struct...end` для описания модулей
- конструкции `signature` и `sig...end` для определения сигнатур модулей
- конструкции `type`, `datatype` для описания новых типов
- конструкция `case ... of ... | ...`
- конструкция описания исключения `exception` и функция создания исключения `raise`
- функции для работы со случайными величинами `Random.rand` и `Random.randRange`
- конструкция для создания функционального значения `fn ... => ...`
- функции для агрегирования значений элементов списка `foldr`, `foldl`
- функция для обработки элементов списка `map`
- функция вычисления длины списка `length`
- функции работы со списками `List.rev`, `List.revAppend`
- функции библиотеки `List`: фильтрации списка `List.filter`; поиска элемента `List.find`
- операция композиции функций `o`
- функции работы со строками `^`, `String.concat`, `String.concatWith`.
- функция сортировки списка `ListMergeSort.sort`.
- функция вывода строки на экран `print`.
- функция ввода строки `TextIO.inputLine`.
- функция чтения числа из строки `Int.fromString`.

6.2. Ограничения

При выполнении данной лабораторной работы нужно соблюдать следующие ограничения:

- При описании функций НЕ ДОЛЖНО БЫТЬ ЯВНЫХ УКАЗАНИЙ ТИПОВ аргументов и результата;
- Все рекурсивные функции должны быть реализованы только как функции с хвостовой рекурсией;
- Компиляция решения не должна приводить к появлению ошибок и предупреждений. Единственное допустимое предупреждение «`Warning: binding not exhaustive`» (может возникать при разложении по шаблону в конструкции `val`). Появление других предупреждений снизит оценку за всю работу на 50%.
- Операции сравнения на равенство `=` и неравенство `<>` можно использовать только для сравнения чисел и строк;
- Нельзя использовать функции, не перечисленные в разделе 6.1. Если вы считаете, что для выполнения какого-то из заданий необходима особая функция или конструкция — задайте вопрос на форуме «Лабораторная работа №3»;

Все разрабатываемые вспомогательные функции должны быть только локальными.

Ограничения на использование конструкций языка не касаются тестов. В файле с тестами можно использовать любые средства.

6.3. Предостережения насчёт решения

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

При реализации функции по шаблонам или использовании конструкции `case` для сравнения с шаблонами необходимо стараться реализовать как можно меньше вариантов ветвления.

При возможности определения функции по шаблонам следует отдавать предпочтение этому способу определения функции.

Следует отдавать предпочтение конструкции `case` перед условной конструкцией (по мере разумного).

Избегайте лишнего оборачивания функции в замыкание. Там где вместо конструкции `fun` можно применить конструкцию `val` следует отдавать предпочтение последней. Там, где можно обойтись без составления лямбда-выражений, следует обойтись без них.

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.

Не следует делать предположений насчёт задания, не сформулированных явно в условии. Если возникают сомнения — задайте вопрос на форуме «Лабораторная работа №3».

7. Задания

1. Опишите каррированную функцию `precedes f lst a1 a2`, проверяющую, расположен ли элемент `a1` в списке `lst` не позже, чем элемент `a2`. Параметр `f` — функция сравнения на равенство значений типа, к которому относятся `a1`, `a2` и элементы списка `lst`.

   Например, вызов

   ```
   precedes (op =) [1, 2, 3, 4, 5] 4 5
   ```

   должен возвращать `true` (значение 4 расположено в списке раньше значения 5).

   Сигнатура итоговой функции: `('a * 'b -> bool) -> 'a list -> 'b -> 'b -> bool`.

   Типовое решение — 4 строки кода.

2. Опишите две каррированные функции `colorIsLT` и `rankIsLT`.

   Функция `colorIsLT` получает на вход две масти и решает расположен ли первый аргумент в списке мастей `colors` не позже чем второй аргумент. Таким образом, если список `colors` задаёт нам порядок мастей «по возрастанию», то функция `colorIsLT` выдаёт `true` если первый цвет «не больше», чем второй.

   Сигнатура функции `colorIsLT` — `color -> color -> bool`.

   Функция `rankIsLT` получает на вход два значения ординарной карты и решает расположен ли первый аргумент в списке `ranks` не позже чем второй аргумент. Таким образом, если список `ranks` задаёт нам порядок значений карт «по возрастанию», то функция `rankIsLT` выдаёт `true` если первый цвет «не больше», чем второй. Сигнатура функции — `rank -> rank -> bool`.

   Обе функции должны быть реализованы с использованием функции `precedes`.

   Примеры вызовов функций:

   ```
   - colorIsLT BLUE GREEN;
   val it = true : bool
   - rankIsLT (NUM 3) SKIP;
   val it = true : bool
   ```

   Решение для каждой функции — 1 строка.

3. Опишите функцию `isLT`, принимающую на вход пару карт и выдающую `true`, если первая карта «не больше», чем вторая. При установлении порядка среди карт руководствуемся следующими принципами:

   - любая дикая карта «больше» любой ординарной карты;
   - карта `WILD_DRAW_FOUR` больше чем `WILD`;
   - отношение между двумя ординарными картами определяется их мастями, а при равенстве мастей — значениями (рангами).

   Например, вызов

   ```
   isLT (WILD, CRD (NUM 3, BLUE))
   ```

   должен возвращать `false` (так как карта `WILD` старше, чем любая ординарная карта).

   Сигнатура функции — `card * card -> bool`.

   Объем решения — 3–4 строки.

4. Опишите функцию `cardSort`, принимающую на вход список карт. Функция должна выдавать исходный список, отсортированный в порядке невозрастания.

   Для реализации решения следует использовать функцию `ListMergeSort.sort` и функцию, полученную в результате решения предыдущего задания.

   Например, вызов

   ```
   cardSort [ CRD (NUM 2, YELLOW), CRD (NUM 5, GREEN), CRD (NUM 8, YELLOW), CRD (REVERSE, YELLOW)
            , WILD, CRD (SKIP, RED), CRD (DRAW_TWO, GREEN) ]
   ```

   должен вернуть список

```
[ WILD ,CRD (SKIP, RED), CRD (DRAW_TWO, GREEN), CRD (NUM 5, GREEN)
, CRD (REVERSE, YELLOW), CRD (NUM 8, YELLOW), CRD (NUM 2, YELLOW)]
```

Сигнатура функции — `card list -> card list`.

Объем решения — 1 строка.

5. Опишите функцию `cardValue`, принимающую карту в качестве аргумента и выдающую ее стоимость. Стоимость каждой карты описывается в разделе 3.1.

   Например, вызов

   ```
   cardValue (CRD (DRAW_TWO,GREEN))
   ```

   должен возвращать `20`.

   Сигнатура функции — `card -> int`.

   Объем типового решения — 3 строки.

6. Опишите функцию `sumCards`, получающую список карт в качестве аргумента и возвращающую сумму очков по данному списку.

   Например, вызов

   ```
   sumCards [ CRD (NUM 2, YELLOW), CRD (NUM 5, GREEN), CRD (NUM 8, YELLOW), CRD (REVERSE, YELLOW)
            , WILD, CRD (SKIP, RED), CRD (DRAW_TWO, GREEN) ]
   ```

   должен вернуть `125`.

   В реализации решения должна использоваться функция, полученная в предыдущем задании.

   Сигнатура функции — `card list -> int`.

   Объем типового решения — 1–5 строк.

7. Опишите функцию `oppositeDir`, противоположное направление хода игры для заданного.

   Например, вызов

   ```
   oppositeDir CLOCKWISE
   ```

   должен вернуть `COUNTERCLOCKWISE`.

   В реализации решения должна использоваться функция, полученная в предыдущем задании.

   Сигнатура функции — `direction -> direction`.

   Объем типового решения — 2 строки.

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 поднимать исключение `e`.

   Например, вызов

   ```
   removeCard ( [ CRD (NUM 2, YELLOW), CRD (NUM 5, GREEN), CRD (NUM 8, YELLOW), CRD (REVERSE, YELLOW)
                , WILD, CRD (SKIP, RED), CRD (DRAW_TWO, GREEN) ]
              , CRD (NUM 5, GREEN)
              , List.Empty)
   ```

   должен вернуть список

   ```
   [ CRD (NUM 2, YELLOW), CRD (NUM 8, YELLOW), CRD (REVERSE, YELLOW)
   , WILD, CRD (SKIP,RED), CRD (DRAW_TWO,GREEN)]
   ```

   В реализации решения должна использоваться функция `isSameCard` для сравнения карт на равенство. Кроме того, для реализации будет полезна функция `revAppend` модуля `List`.

   Сигнатура функции — `card list * card * exn -> card list`.

   Объем решения составит порядка семи строк.

9. 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 3.1.

   Например, вызов

   ```
   cardCount WILD
   ```

   должен вернуть `4`.

   Сигнатура функции — `card -> int`.

   Объем решения — 3–4 строки.

10. Создайте переменную `deck`, которой присвойте вычисленный список — колоду из 108 карт для игры в «Uno». В качестве заготовок для вычислений стоит использовать списки `colors` и `ranks`, заданные в в файле `lab-3-use.sml`.

    Вычисление списка карт следует начать с построения списка, в который каждая карта включена 1 раз: это список полученный в результате создания карты для каждого элемента «декартова произведения» списков `ranks` и `colors`, к которому добавлены по экземпляру карт `WILD` и `WILD_DRAW_FOUR`. На основе построенного списка следует получить колоду карт, преумножив в нем каждую карту нужное количество раз (функция `cardCount` вернёт для карты число, которое она должна повторяться, а `addNCopies` поможет добавить нужное количество в результат).

    Типовое решение использует функции `map`, `foldr`, `@` и лямбда-выражения.

    Тип переменной `deck` — `card list`.

    Типовое решение — 7 строк.

11. Опишите функцию `deal`, раздающую карты игрокам. Функция должна получать в качестве аргумента список игроков (список типа `player list`), задающий порядок расположения игроков по часовой стрелке, начиная с раздающего.

    В начале своей работы функция должна вывести на экран сообщение о том, кто является раздающим (заготовка для вывода сообщения приводится в комментарии шаблона решения).

    Результатом функции должна быть конфигурация игры сразу после раздачи (элемент типа `desk`). Функция должна сначала перетасовать целую колоду карт, после чего выделить каждому игроку 7 карт из колоды. Полученный набор элементов типа `player` составит итоговый 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`, направление хода игры — `CLOCKWISE`, и «память» — пустой список.

    Для решения нужно описать вспомогательную функцию, организующую итерационный процесс обработки заданного списка. Следует использовать функцию `pass` для перекладывания из колоды в список карт игрока необходимого количества карт. Кроме того, типовое решение использует функцию `rev` модуля `List`. Понятно, что потребуется использовать конструктор для элемента типа `desk`, геттеры и сеттеры для элементов типа `player`.

    Задача данной функции — раздать игрокам карты. Элемент типа `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 займётся другая функция (задание 13).

    Сигнатура функции — `player list -> desk`.

    Типовое решение — 15 строк.

12. Опишите функцию `nextPlayer`, получающую и возвращающую элемент типа `desk`. Функция должна описать передачу хода от одного игрока к следующему. Реализация функции должна заключаться в перестановке элементов в списке игроков в зависимости от направления хода игры. Если игра ведётся по часовой стрелке, то следующим игроком является тот, который в исходном списке игроков находится на второй позиции, а тот, который был первым, должен встать на последнюю позицию в списке (циклический сдвиг влево). Если игра ведётся против часовой стрелки, то следующим по ходу игры должен быть игрок, находящийся в исходном списке на последней позиции: его нужно переместить на первую позицию в списке (циклический сдвиг вправо).

    После формирования новой конфигурации игры с обновлённым списков игроков функция должна вывести на экран имя очередного игрока с помощью функции `Desk.whoseTurn`.

При реализации функции будет полезна функция `rev` а так же сеттеры и геттеры модуля `Desk`.

Сигнатура функции — `desk -> desk`.

Типовое решение — 16 строк.

13. Опишите функцию `deskNormalize`, 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 (поле `direction`) 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 3.2.

    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`.

    В случае, когда верхняя карта списка `pile` — карта `REVERSE`, функция, при помощи вызова `Desk.whoseTurn`, должна вывести на экран имя игрока, который должен сделать первый ход.

    Решение должно использовать функции `nextPlayer`, `shuffleList`, а так же геттеры и сеттеры модуля `Desk`.

    Сигнатура функции — `desk -> desk`.

    Объем типового решения — 14 строк.

14. Опишите функцию `draw`, 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`, а так же геттеры и сеттеры модуля `Desk`.

    Сигнатура функции — `desk -> desk`.

    Объем типового решения — порядка 16 строк.

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

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

    Сигнатура функции — `desk -> desk`.

    Объем типового решения — порядка 1 строка.

16. Опишите функцию `drawFour`, 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 `draw` использовать функцию `drawTwo`.

    Сигнатура функции — `desk -> desk`.

17. Опишите каррированную функцию `playableWhenWild`, определяющую, может ли игрок сделать ход указанной картой, когда на колоде «Сброс» лежит дикая карта и заказан некоторый цвет. Функция принимает в качестве первого аргумента значение заказанного цвета, а вторым аргументом — карту, о которой нужно сделать заключение. О том, какими картами игрок может ходить в заданной ситуации описано в разделе 3.3.

    Например, вызов

    ```
    playableWhenWild BLUE WILD_DRAW_FOUR
    ```

    должен вернуть `true`.

    В решении задания нужно использовать функцию `isSameColor`.

    Сигнатура функции — `color -> card -> bool`.

    Объем решения — 2 строки.

18. Опишите каррированную функцию `playableWhenExec`, определяющую, может ли игрок сделать ход указанной картой, когда на колоде «Сброс» лежит активная карта и состояние хода `EXECUTE`. Функция принимает в качестве первого аргумента значение активной карты, а вторым аргументом — карту, о которой нужно сделать заключение. О том, какими картами игрок может ходить в заданной ситуации описано в разделе 3.3.

    Например, вызов

    ```
    playableWhenExec SKIP WILD_DRAW_FOUR
    ```

    должен вернуть `false`.

    В решении задания нужно использовать функцию `isSameRank`.

    Сигнатура функции — `rank -> card -> bool`.

    Объем решения — 2 строки.

19. Опишите каррированную функцию `playableWhenProc`, определяющую, может ли игрок сделать ход указанной картой, когда на колоде «Сброс» лежит ординарная карта и состояние хода `PROCEED`. Функция принимает в качестве первого аргумента пару из ранга и масти карты на колоде «Сброс», а вторым аргументом — карту, о которой нужно сделать заключение. О том, какими картами игрок может ходить в заданной ситуации описано в разделе 3.3.

    Например, вызов

    ```
    playableWhenProc (NUM 5, RED) (CRD (SKIP, RED))
    ```

    должен вернуть `true`.

    В решении задания нужно использовать функции `isSameColor` и `isSameRank`.

    Сигнатура функции — `rank * color -> card -> bool`.

    Объем решения — 3 строки.

20. Опишите функцию `cardsToPlay`, 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.

    В решении задания нужно использовать функции `List.filter`, `playableWhenWild`, `playableWhenExec`, `playableWhenProc`.

    Сигнатура функции — `desk -> card list`.

    Объем форматированного типового решения — 11 строк.

21. Опишите функцию `requiredColor`, получающую элемент типа `desk` в качестве аргумента. Функция должна выдавать цвет, который определяется текущей конфигурацией игры. Если наверху колоды сброс дикая карта, то результат функции — заказанный цвет, а в противном случае результат функции — цвет карты наверху колоды «Сброс».

    Сигнатура функции — `desk -> color`.

    Типовое решение — 5 строк.

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

    Например, вызов

    ```
    hasColor ( RED
             , [ CRD (NUM 2, YELLOW), CRD (NUM 5, GREEN), CRD (NUM 8, YELLOW), CRD (REVERSE, YELLOW)
               , WILD, CRD (SKIP, RED), CRD (DRAW_TWO, GREEN) ]
             )
    ```

    должен вернуть `true`.

    Решение заключается в использовании функций `isSome` и `List.find`.

    Сигнатура функции — `color * card list -> bool`.

    Решение — 7 строк.

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

    Решение состоит в том, чтобы извлечь список игроков, после чего для каждого игрока извлечь список его карт и подсчитать в нем количество элементов.

    Сигнатура функции — `desk -> int list`.

    Объем решения — 1–2 строки.

24. Опишите модуль `Naive`, ограниченный сигнатурой `STRATEGY`. В модуле определите единственную функцию `ownStrategy` тип которой соответствует типу `strategy`, т.е. функция принимает на вход 7 аргументов:
    - состояние текущего хода;
    - полный список карт игрока;
    - список карт игрока, которыми возможно сделать ход в текущей конфигурации игры;
    - карту, лежащую наверху колоды «Сброс»;
    - направление хода игры;
    - список последних ходов в игре;
    - список целых чисел, в котором каждое число — количество карт на руках игрока. Элементы следуют в списке в порядке рассадки игроков, начиная с текущего игрока.

    Результат функции `ownStrategy` этого модуля должен зависеть только от значения третьего аргумента: функция должна проводить сортировку списка карт (в порядке убывания), полученного в качестве третьего аргумента и делать ход картой, являющейся головой отсортированного списка. В случае, если первая карта отсортированного списка дикая, то очередной ход — заказ (`ORDER`) выбранного случайно цвета. Если же первая карта ординарная, то стратегия должна делать ординарный ход (`SIMPLE`).

    Для сортировки списка следует использовать функцию `cardSort`. Чтобы выбрать случайный цвет, следует перемешать список `colors` и взять первый элемент получившегося списка.

    Например, вызов

```
Naive.ownStrategy ( PROCEED
                  , [ CRD (NUM 2, YELLOW), CRD (NUM 5, GREEN), CRD (NUM 8, YELLOW), CRD (REVERSE, YELLOW)
                    , CRD (SKIP, RED), CRD (DRAW_TWO, GREEN), CRD (DRAW_TWO, BLUE) ]
                  , [CRD (NUM 5, GREEN), CRD (SKIP, RED)]
                  , CRD (NUM 5, RED)
                  , CLOCKWISE
                  , [SIMPLE (CRD (NUM 5, RED))]
                  , [7, 7, 6]
                  )
```

    должен вернуть `SIMPLE (CRD (SKIP,RED))`.

    Типовое решение — 11 строк.

25. Опишите функцию `moveAction`, вносящую в конфигурацию игры реакцию на ход игрока. Функция должна получать в качестве аргумента кортеж `(mv, dsk, hasColor)`, в котором `mv` — ход игрока (значение типа `move`), `dsk` — конфигурация игры сразу после того, как игрок выложил свою карту в колоду «Сброс» (значение типа `desk`), и значение `hasColor` — отметка о том, имелась ли на момент хода на руках у игрока карта того цвета, который требовался в конфигурации игры (значение типа `bool`).

    Функция предполагает, что игрок выбрал некоторый ход и карта, которой он пошёл, уже переложена из его рук на колоду «Сброс» и ход зафиксирован в «памяти» конфигурации. Теперь необходимо в зависимости от этого хода поменять остальные параметры конфигурации и передать ход игроку следующему по очереди.

    Изменения следует производить руководствуясь следующими правилами:
    - если очередной ход — `ORDER (WILD_DRAW_FOUR, col)`, где `col` — некоторый цвет, то следует установить состояние очередного хода `GIVE col` и проверить, отсутствуют ли у игрока карты того цвета, который требовался для хода (значение, переданное функции третьим аргументом). Если карты нужного цвета у игрока были, то следует отсчитать игроку 4 карты из колоды «Прикуп» и передать ход следующему игроку. В противном случае, отсчитать 4 карты очередному игроку, записать в «память», что он пропускает ход (`PASS`), и передать ход игроку, следующему за ним.
    - если очередной ход — `ORDER (WILD, col)`, где `col`, то следует установить состояние очередного хода `GIVE col` и передать ход следующему игроку.
    - если очередной ход — `SIMPLE (REVERSE, col)`, где `col` — некоторый цвет, то следует установить состояние очередного хода `PROCEED`, поменять направление хода игры на противоположное и передать ход следующему игроку.

- если ход сделан активной картой, отличной от `REVERSE`, то следует установить состояние очередного хода `EXECUTE` и передать ход следующему игроку
- если ход сделан цифровой картой, то следует установить состояние хода `PROCEED` и передать ход следующему игроку.

Для решения потребуются функции, описанные ранее: `oppositeDir`, `nextPlayer`, а так же сеттеры и геттеры модуля `Desk`.

Сигнатура функции — `move * desk * bool -> desk`.

Форматированное типовое решение — 19 строк.

26. Опишите функцию `askPlayerForCard`, получающую игрока `plr` и кортеж `args`, представляющий собой аргументов для функции стратегии. Функция должна спросить функцию стратегии игрока об очередном ходе при заданных аргументах `args`.

    Сначала нужно узнать, управляется ли игрок вручную. Если это не так, то получить от игрока его функцию стратегии и спросить её результат от набора аргументов `args`.

    Если же игрок управляется вручную, то спросить об очередном ходе следует функцию стратегии `Manual.ownStrategy` (модуль `Manual` приведён в закомментированном виде в шаблоне решения). Если функция `Manual.ownStrategy` вернёт не `PASS`, то полученный ход и нужно принять за ход игрока. Если же функция ручной стратегии вернёт `PASS`, то следует у игрока изменить режим управления на автоматический и переспросить про очередной ход у функции стратегии игрока.

    Результат выполнения функции — пара составленная из игрока (возможно, сменившего режим управления) и его хода.

    Решение должно использовать геттеры и сеттеры модуля `Player`.

    Сигнатура итоговой функции —
    ```
    player * (state * card list * card list * card * direction * move list * int list)
    -> player * move
    ```
    .

    Решение — порядка 6 строк.

27. 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 запросить ход игрока с помощью `askPlayerForCard`. Получив результат — ход игрока, функция должна выполнить действие, соответствующее одному из следующих условий:

    - если ход игрока не является допустимым (карта, выбранная игроком отсутствует в списке `playCards` или ход игрока `PASS`), то функция должна создавать исключение `IllegalMove (name, args)`, где `name` — имя игрока, сделавшего ход, а `args` — кортеж из аргументов, переданных функции стратегии игрока;

    Во всех остальных случаях нужно

    - вывести на экран сообщение о выбранном ходе игрока (заготовка для вывода сообщения приводится в комментарии шаблона решения);
    - удалить из карт игрока выбранную карту и поместить ее наверх колоды «Сброс»;
    - добавить в «память» информацию о ходе игрока;
    - скорректировать конфигурацию игры с помощью функции `moveAction`, которой передать в качестве 3-го аргумента ответ на вопрос о том, были ли у игрока на руках карты того цвета, который требовался для хода перед тем, как игрок выбрал карту (результат вызова `requairedColor`).

    Кроме уже названных функций для решения потребуются функции, описанные ранее: `countCards`, `removeCard`, `requiredColor`, `hasColor`.

    Сигнатура функции — `desk * card list -> desk`.

    Форматированное типовое решение — 32 строки.

28. Опишите функцию `execution`, получающую и возвращающую элемент типа `desk` в котором наверху колоды «Сброс» лежит активная карта и состояние хода `EXECUTE`. Функция должна моделировать ситуацию, когда у очередного игрока нет карты, которой можно было бы сделать ход.

Прежде всего нужно внести в «память» информацию, что очередной игрок пропускает ход (его ход `PASS`).

Функция может рассматривать следующие ситуации:

- 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`;
- других ситуаций быть не может.

Для решения стоит использовать функции `nextPlayer`, `drawTwo`.

Сигнатура функции — `desk -> desk`.

Объем форматированного типового решения — 10 строк.

29. Опишите функцию `yourTurn`, 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 «понести наказание» (с помощью функции `execution`). Для других состояний — игрок должен взять одну карту из колоды «Прикуп» (с помощью функции `draw`), и если список карт, которыми можно сделать ход, останется пустым — игрок должен пропустить ход (передав ход следующему игроку), оставив отметку в «памяти» о пропущенном ходе (ход `PASS`), в противном случае — делать ход.

    Для решения стоит использовать функции `nextPlayer`, `cardsToPlay`, `draw`, `execution`, `play`.

    Сигнатура функции — `desk -> desk`.

    Типовое решение — 12 строк.

30. Опишите функцию `playerLoss`, получающую значение типа `player`. Для заданного игрока функция должна выдавать пару из имени игрока и суммарной стоимости карт у него на руках.

    Для решения следует использовать функцию `sumCards`.

    Сигнатура функции — `player -> string * int`.

    Типовое решение — 1–3 строки.

31. Опишите функцию `checkWinner`, получающую элемент типа `desk` в качестве аргумента и проверяющую, найдётся ли в списке игроков текущей конфигурации победивший игрок, т.е. игрок у которого список карт пуст. Функция должна возвращать `NONE`, если победитель отсутствует. Если же такой игрок найдётся, то функция должна вернуть обёрнутый в контейнер `SOME` кортеж из имени победителя и списка пар имя-проигрыш, составленных для каждого игрока списка игроков.

    Типовое решение использует функции `null`, `List.find`, `List.map`, геттеры модулей `Desk` и `Player` и операцию композиции `o`.

    Сигнатура функции — `desk -> (string * (string * int) list) option`.

    Объем типового решения — 9 строк.

32. Опишите функцию `game`, получающую список игроков в качестве аргумента. 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.

    Функция должна раздать игрокам карты, затем нормализовать начальную конфигурацию игры. После этого должен запускаться итерационный процесс: выполнять функцию `yourTurn` до тех пор, пока не появится игрок с пустым списком карт. Как только появится победитель, итерационный процесс должен остановиться и должен быть извлечён результат функции.

    Для реализации решения понадобятся функции `deal`, `deskNormalize`, `yourTurn`, `checkWinner`.

    Форматированное решение — 9 строк.

Общий объем решения, включая строки, предварительно заданные в файле, должен составить порядка 500 строк.