CS 583 - Final Report: Uno design and implementation

Authors:

- Shujin Wu: `932-293-747`
- Yuan Zhang: `932-094-862`
- Yipeng Wang: `931-903-609`

Project Overview

Uno is a typical card game which is popular in America, so domain is the card game. Obviously, users are uno players. In this project, users can play uno with `1-4` robot players, simulating the actions, such as draw/drop cards, pick a color/reverse direction, declare uno, etc. We didn't find any playable uno implemented in Haskell.

For more information about Uno, please visit the website `https://en.wikipedia.org/wiki/Uno_(card_game)`

Implementation

The following session lists the most important types, type classes as well as the functions in the project.

- types - `UnoDataModels.hs`
  - **Game**:Use transformer to combine State monad and IO monad.
  - **GameState**: Abstract the game related states, such as player states, deck states, whose turn, etc.
  - **PlayerState**: Abstract the player related states.
  - **Card**: Describe card information.
- functions
  - **uno**: Game entry. It will initialize the game state based on number of players.
  - **simGame**: Set the game starting status. It will deal cards for player, decide the starting player, starting card and direction.
  - **goPlay**: Players play game in turns.
  - **drawCards**: Designated player draws designated number of cards to his/her hand deck.
  - **cardPile**: Initialize the deck.
  - **dropCard**: Designated player drops a designated card. This function will show which card are droped, update game state and run the card effect.
  - **runEffect**: There are six types of cards in Uno, so we implemented six card effects accordingly. runEffect will run the card effect based on card type.
  - **getLineInt**: Get user input with error handlling

- **helpInfo**: game helper; users can search for rules at any time.

Design Decisions

The following lists some important design decisions we made during the implementation.

Design desition of type: Card

Our Card type has experienced most refactorings during the implementation. In Uno, there are six types of cards, Regular card has both color and number propertities; function cards(Skip, DrawTwo, Reverse) only have color; Wild and WildDrawFour have no color and number.

Therefore, our very first version of Card is defined as below:

```
data Card = Skip Color
          | DrawTwo Color
          | Reverse Color
          | Wild
          | WildDrawFour
          | Regular Color Int
```

Each card type has its own effect, then we decided to use the higher-order type within the Card type, and use the Maybe to handle the error.

```
data CardType = Skip
              | DrawTwo
              | Reverse
              | Wild
              | WildDrawFour
              | Regular
              deriving(Show)

data Card = Card { num :: Maybe Int
                 , clr :: Maybe Color
                 , effect :: GameState -> IO GameState
                 , cardType :: CardType
                 , desc :: String
}
```

Maybe makes pattern match tedious and long, so we decided to add another color as no_color. There were a couple attempts we made to try to run the card effects, such as use the Data.Map type for cardType, our thinking is making it easier for indicating the cardType for effects. However, it turns out didn't work.

```
data Card = Card { num :: Maybe Int
                 , clr :: Color
                 , effect :: GameState -> IO GameState
```

```
                   , cardType :: M.Map Int CardType
                   , desc :: String
}
```

Finally, we took effect out of Card type, and created a new function to resolve the problem. We also deriving Eq for Card, making it easier for Card comparison.

```
data Card = Card { num :: Int
                 , clr :: Color
                 , cardType :: CardType
                 , desc :: String
} deriving(Eq)

data CardType = Skip
              | DrawTwo
              | Reverse
              | Wild
              | WildDrawFour
              | Regular
              deriving(Show, Eq, Enum)

runEffect :: CardType -> Game ()
runEffect _cardType = case _cardType of
                          Regular       -> regular
                          Skip          -> skip
                          Reverse       -> reverseD
                          DrawTwo       -> drawTwo
                          Wild          -> wild
                          WildDrawFour  -> wildDrawFour
```

Design decision of type: PlayerState

The most important decision we made for PlayerState is to instanciate Eq, Ord for it. After that, we can easily take advantage of Data.List features, for instance, we can use the `minimum` function to immediately take the winner out of player list. If without Ord instance, `getWinnerId` will be a lot more complecate.

```
data PlayerState =  PlayerState{
      pId :: Int
    , name :: String
    , score :: Int
    , cardsInHand :: [Card]
} deriving(Show)

instance Eq PlayerState where
  _player1 == _player2 = score _player1 == score _player2

instance PlayerState where
  _player1 <= _player2 =  score _player1 <= score _player2
  _player1 > _player2  =  score _player1 > score _player2
```

```
-- @Int Winner ID
getWinnerId :: [PlayerState] -> Int
getWinnerId _players = pId $ minimum _players
```

Design decision of type: Game

In the ealier version, we use `IO GameState` through out the game, so we would need to parse in/out gameState whenever we need to interact with game state.

```
data GameState = GameState {
      dir :: Direction
    , realPlayer :: Int
    , whoseTurn :: Int
    , currCard :: Card
    , players :: [PlayerState]
    , deck :: [Card]
} deriving(Show)

skip :: GameState -> IO GameState
skip game@GameState{whoseTurn=_whoseTurn,players=_players,dir=_dir} = return
game{whoseTurn=_nextTurn}
   where _nextTurn = getNextTurn (getNextTurn _whoseTurn _players _dir) _players
_dir

drawTwo ::GameState -> IO GameState
drawTwo  game@GameState{whoseTurn=_whoseTurn,players=_players,dir=_dir} = do
    game' <- drawCards 2 game _nextTurn
    return game'{whoseTurn=_nextTurn}
  where
    _nextTurn = getNextTurn _whoseTurn _players _dir
```

In the above version, it's hard to break code into sub-steps. In the stateT version, since we can get game state whenever we need, and we don't need to parse it in explicitly, it's easier to break the process into substeps then reuse it. The code is so much clearer as you see below.

```
data GameState = GameState {
      dir :: Direction
    , realPlayer :: Int
    , whoseTurn :: Int
    , currCard :: Card
    , players :: [PlayerState]
    , deck :: [Card]
    , isOver :: Bool
    , ithTurn :: Int
} deriving(Show)

type Game = StateT GameState IO

skip :: Game ()
skip = setNextTurn 2
```

```haskell
drawTwo :: Game ()
drawTwo = do
  setNextTurn 1
  game <- get
  modify $ drawCards 2 $ whoseTurn game
  setNextTurn 1
```