Emurgo Acadamy Project

Zanzibar Dice Game

What we need to play the Zanzibar ?

- Two or more players
- Three six-sided dice
- Token (counter or chips or ..)

How do you play the Zanzibar ?

The first player may roll the dice up to three times in an attempt to get as high a score as possible. (See How do you score? below.) They may stop rolling after the first or second roll if they wish.
The other players, in turn, then try to roll a higher score in the same number or fewer rolls than the first player.
Once all players have had a turn, the player with the lowest score receives a number of token from the other players. The number of token they receive depends on the hand of the other players.

How to score ?

The highest ranking combinations are shown in descending order:

- 4,5,6 - Zanzibar
- 1,1,1
- 2,2,2
- 3,3,3
- 4,4,4
- 5,5,5
- 6,6,6
- 1,2,3

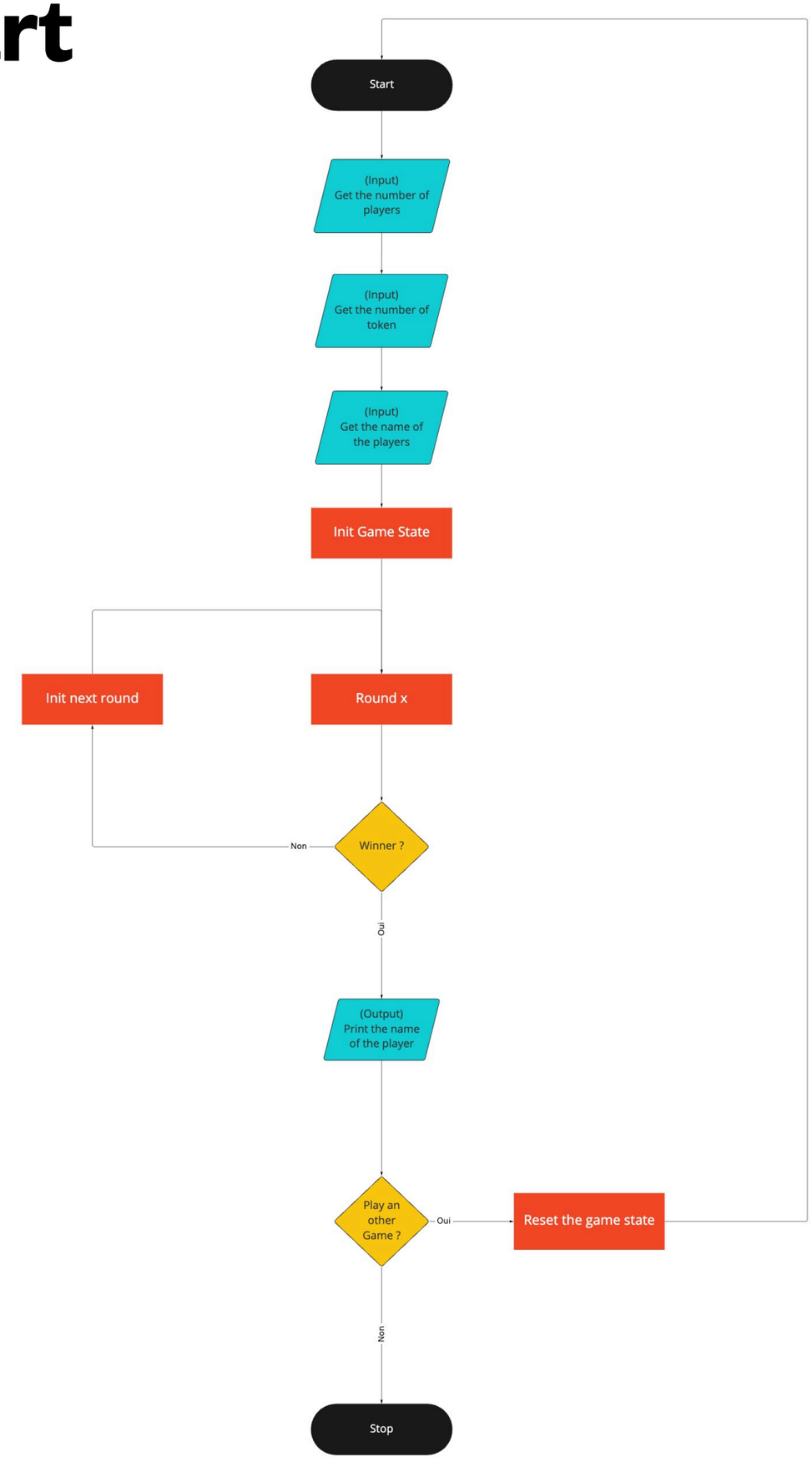
All other combinations rank as a sum of the three dice added together:

- 1 = 100 points
- 6 = 60 points
- 2 = 2 points
- 3 = 3 points
- 4 = 4 points
- 5 = 5 points

Token distribution

- 1 chip if the player score a points total
- 2 chips if the player score 1,2,3
- 3 chips if the player has a combination (Three same number)
- 5 chips if the player has the combination 4,5,6 (Zanzibar)

Game flow chart



miro

Score implementation (Types)

```haskell
data DiceNb
    = One
    | Two
    | Three
    | Four
    | Five
    | Six
    deriving (Show)

data Rolls
    = Rolls [DiceNb] | NoRolls deriving (Show)
```

Score implementation (Instance Eq, Ord)

```haskell
instance Eq Rolls where
    (==) a b =
        let
            rolls NoRolls                           = 0
            rolls (Rolls [One, One, One])           = 1
            rolls (Rolls [Two, Two, Two])           = 2
            rolls (Rolls [Three, Three, Three])     = 3
            rolls (Rolls [Four, Four, Four])        = 4
            rolls (Rolls [Five, Five, Five])        = 5
            rolls (Rolls [Six, Six, Six])           = 6
            rolls rolls                             = calcScore rolls
        in
            (rolls a) == (rolls b)
```

```haskell
class CalcScore a where
    calcScore :: (Fractional b) => a -> b

instance CalcScore Rolls where
    calcScore (Rolls [a, b, c]) =
        let score One     = 100
            score Two     = 2
            score Three   = 3
            score Four    = 4
            score Five    = 5
            score Six     = 60
        in (score a) + (score b) + (score c)
```

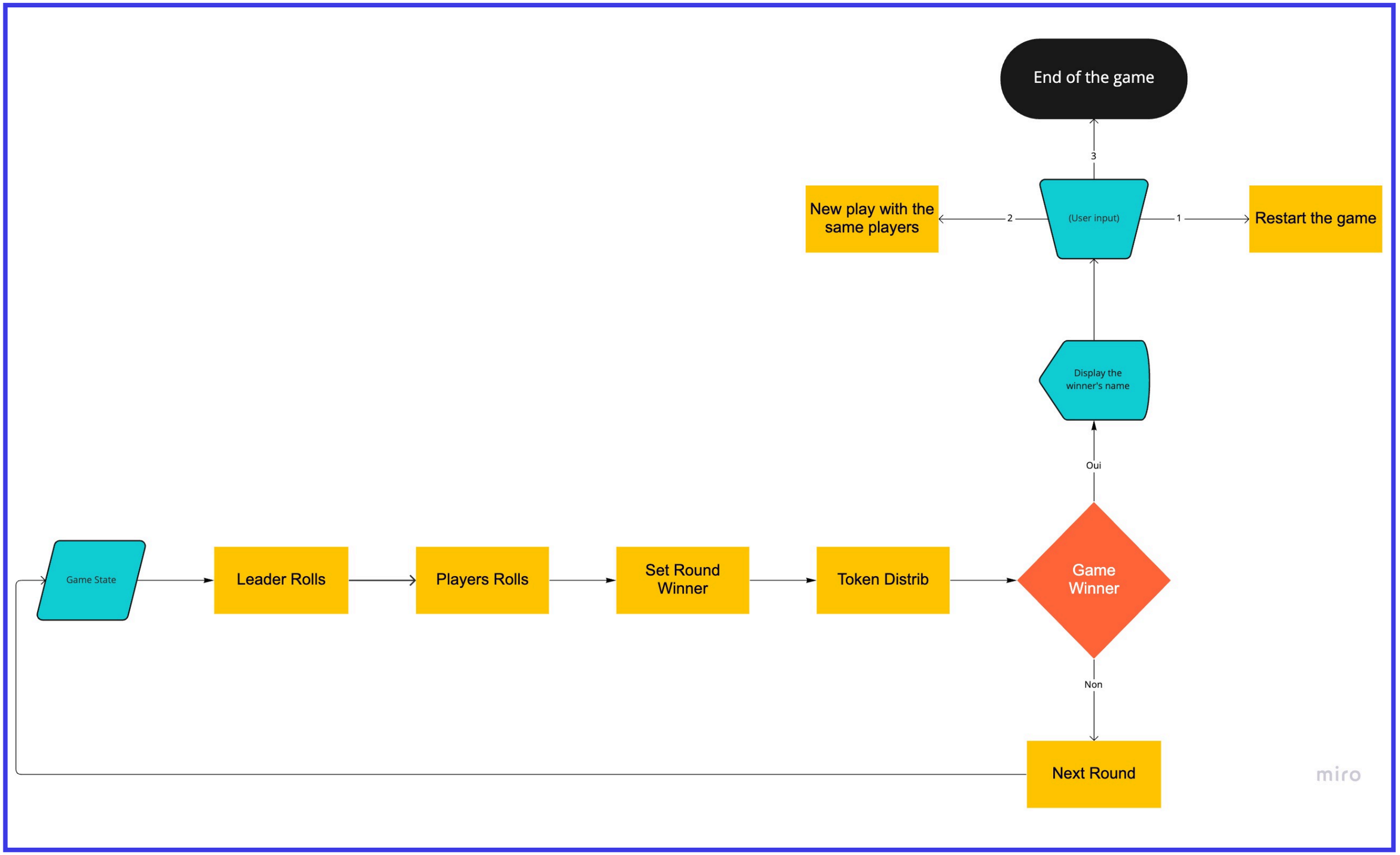
```haskell
instance Ord Rolls where
    compare a b =
        let
            rolls NoRolls                       = 0
            rolls (Rolls [Four, Five, Six])     = 8 -- Zanzibar
            rolls (Rolls [One, One, One])       = 7
            rolls (Rolls [Two, Two, Two])       = 6
            rolls (Rolls [Three, Three, Three]) = 5
            rolls (Rolls [Four, Four, Four])    = 4
            rolls (Rolls [Five, Five, Five])    = 3
            rolls (Rolls [Six, Six, Six])       = 2
            rolls (Rolls [One, Two, Three])     = 1
            rolls rolls                         = (((calcScore rolls) * 99) / 260) / 100 -- (Value > 0 and <= 0.99)
        in
            compare (rolls a) (rolls b)
```

Max score -> 260 (One, One, Six)
Min score -> 9 (Two, Three, Four)

<u>The point total value will be betwen 0.03 and 0.99:</u>

- (260 * 99 / 260 / 100 = 0.99
- (9 * 99 / 260 / 100 = 0.03

Round flow chart



Play Round function

Round implementation (State transformer)

Data types declaration

```haskell
data GameStatus = Winner | NoWinner deriving Show

data GameStates = GameStates
    {
        _players        :: Players,
        _round          :: Int,
        _rLeader        :: RPlayer,
        _scoreboard     :: [RPlayer],
        _nbOfTry        :: Int,
        _rWinner        :: Player,
        _gStatus        :: GameStatus
    }  deriving (Show)
```

Function to play a round

```haskell
playRound :: StateT GameStates IO ()
playRound = do
    leaderRolls
    playersRolls
    setRWinner
    tokenDistrib
    winnerCheck
    printRWinner
```

Function to takes the winner of the last round or round 0 for rolls the dice first.

*Round 0 is the round where all players rolls the dice once to determinate who will rolls first in the round 1.

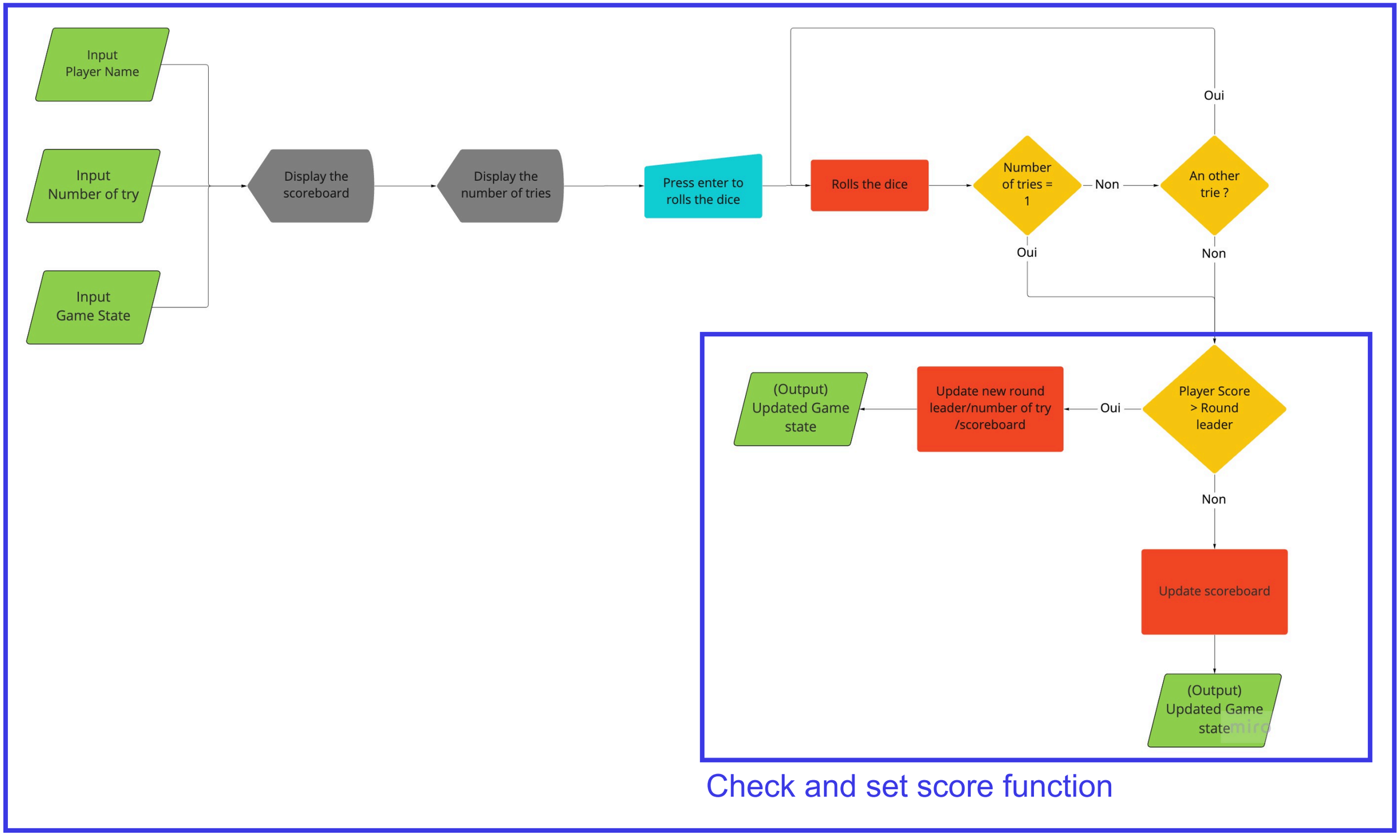
Function to pick a player who have not scored to rolls the dice.
If he has a better score than the leader he become the new leader.
The number of tries is updated if he made a better score with less tries.

Function to set the round leader as round winner.

Function to give to the player with the lower score a certain amount of token from the other player based on their scores.

Function to check if a player have give all his token and so win the game.

Player rolls flow chart



Player rolls functions

```haskell
playerRolls :: Player -> Int -> StateT GameStates IO ()
playerRolls player nbOfTry = do
    s <- get
    lift $ clearScreen
    printScoreBoard
    -- Get the player ID and name of the player.
    let (pid, name, _) = player
    -- Print the number of try than the player have and wait for the player to press enter to roll the dice.
    printNBTryMsg name nbOfTry
    _ <- lift $ getLine
    -- Roll the dice and print the result in the terminal.
    rolls <- lift $ rollThreeDice
    printPCombiMsg name rolls
    -- Player combination
    let pCombi = (pid, rolls)
    -- If the number of try is >= 1, ask if the user want an other try or update the game state.
    case nbOfTry of
        1 -> do
            checkAndSetCombi pCombi 3
        _ -> do
            printLine
            printAnOtherTryMsg name
            answer <- lift $ getAnswer1
            case answer of
                "Y" -> do
                    lift $ clearScreen
                    playerRolls player (nbOfTry - 1)
                "N" -> do
                    printLine
                    checkAndSetCombi pCombi (if nbOfTry == 3 then 1 else nbOfTry)
```

Player rolls function

```haskell
checkAndSetCombi :: RPlayer -> Int -> StateT GameStates IO ()
checkAndSetCombi rPlayer nbTry = do
    s <- get
    let cNbTry = _nbOfTry s
    -- Get the combination of the player.
    let (pID, pCombi)  = rPlayer
    --  Get the combination of the round leader.
    let (lID, lCombi)  = _rLeader s
    -- Get the name of the current player.
    let pName          = getPName pID (_players s)
    -- Check if the current player have a better combination than the round leader.
    case (pCombi > lCombi) of
        True -> do
            setRLeader rPlayer
            setNbOfTry (if cNbTry >= 1 && nbTry == 1 || nbTry == 3 && lCombi /= NoRolls  then 1 else nbTry)
            addPScore rPlayer
            printBCMsg pName lCombi
            playersRolls
        False -> addPScore rPlayer
```

Check and set score function