Real-world Functional Programming

Coursework Part II Report

14274056 Junsong Yang (psyjy3)

December 11, 2019

1 Task II.1

```haskell
data Fork = MkFork (TVar Bool)

newInfoBuf :: IO (TChan String)
newInfoBuf = newTChanIO

newFork :: IO Fork
newFork = do
  fork <- newTVarIO False
  return (MkFork fork)

takeForks :: Fork -> Fork -> STM ()
takeForks (MkFork l) (MkFork r) = do
  isUsedL <- readTVar l
  isUsedR <- readTVar r
  if isUsedL || isUsedR then retry
  else do writeTVar l True
          writeTVar r True

putForks ::  Fork -> Fork -> STM ()
putForks (MkFork l) (MkFork r) = do
  writeTVar l False
  writeTVar r False

hungry :: String -> String
hungry name = name ++ " is hungry."

eating :: String -> String
eating name = name ++ " is eating."

thinking :: String -> String
thinking name =  name ++ " is thinking."

```

(a) Dinning Phhilosopher Part I

```haskell
philosophers :: [String]
philosophers = ["Aristotle", "Kant", "Spinoza", "Marx", "Russel"]


randomDelay :: IO ()
randomDelay = do
  waitTime <- randomRIO (1,3)
  threadDelay (waitTime * 1000000)

putBuf :: TChan String -> String -> STM ()
putBuf buf str = writeTChan buf str

getBuf :: TChan String -> STM String
getBuf buf = do
  str <- readTChan buf
  return str

printBuf :: TChan String -> IO ()
printBuf buf = do
  str <- atomically $ getBuf buf
  putStrLn str
  printBuf buf


dinning :: TChan String -> String -> (Fork, Fork) -> IO ()
dinning buf name (left, right) = forever $ do
  atomically $ putBuf buf (hungry name)
  atomically $ takeForks left right
  atomically $ putBuf buf (eating name)
  randomDelay
  atomically $ putForks left right
  atomically $ putBuf buf (thinking name)
  randomDelay

main = do
  forks <- replicateM 5 newFork
  infoBuf <- newInfoBuf
  let dinningPhil     = map (dinning infoBuf) philosophers
      forkPairs       = zip forks (tail . cycle $ forks)
      withForks = zipWith ($) dinningPhil forkPairs
  mapM_ forkIO withForks
  printBuf infoBuf
```

(b) Dinning Phhilosopher Part II

Figure 1: Dinning Phhilosopher

Figure 1 contains two figures that show the full implementation of dinning philosopher using STM. Starting with figure 1 (a), the Fork type is defined with a constructor called MkFork which take a TVar with a Bool as an input. If the boolean inside the TVar is false then this fork is available to be used otherwise this fork is already taken. The newInfoBuf function will return a TChan with String wrapped in IO monad to be used later to store logs that indicate the running state of each thread. The newFork function will return a Fork wrapped in IO monad with the boolean value set to false.

The next two function takeForks and putForks are related to require and release resources. The takeForks function

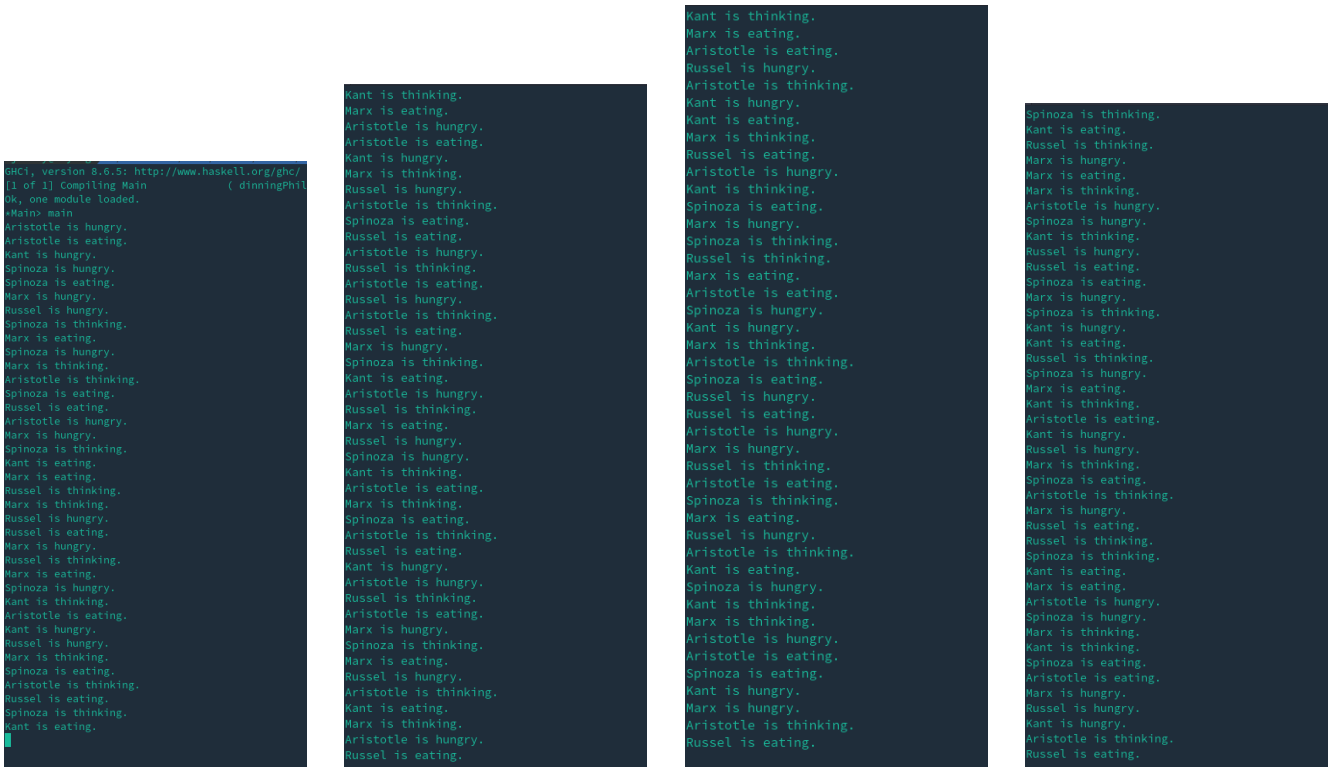
takes two Fork as input. This function will first check if the two Forks are both available. The two Forks can only be used if they are both available at the same time. Otherwise, this function will keep retrying until both Forks can be required. The putForks function is simple just release the two Forks by setting the boolean to true.

The next three functions hungry, eating, thinking are just dummy function that concatenates the name of the philosopher with corresponding information. This information will later be put into the infoBuf(TChan String). The names of philosophers are defined as a list of string as figure 1 (b) shows.

The randomDelay function will call threadDelay to delay the running thread randomly from 1 to 3 seconds. The next three functions putBuf, getBuf printBuf are operations related to the infoBuf(TChan String) that is used to store the logs for running thread. The putBuf function will just store string to the infoBuf(TChan String), the getBuf function will return the string stored in TChan and wrap in STM monad. The printBuf will just print the string stored in TChan.

The dinning function contains the implementation of dining philosophers. This function takes a TChan String(used to store logs), a string(indicates philosopher's names), and a pair of Fork as input. This function will first store a log in the TChan that indicates the philosopher is hungry. Then, this function will try to acquire the Fork using takeForks function. If the forks are acquired successfully, another log will be stored in the TChan that suggests the philosopher is eating. Followed by a random delay from 1 to 3 seconds, the forks will be released using putForks function and the corresponding log will be put into the TChan. Finally, the function ended with another random delay.

The main function will first initialise 5 Forks using newFork function and an infoBuf using newInfoBuf functions. Then the philosophers' name will be bundled with the dinning function using the map function. Then, an infinite list of pair of forks will be generated such that each fork in the pair is distinct from the other. Followed by coupling pairs of forks with the dinning function, a list of runnable functions is made. Finally, the main function will run those function by mapping forkIO function to the runnable dining functions while the printBuf function is called to print the logs of those running thread.



(a) running at 10s (b) running at 1m (c) running at 2m (b) running at 4m

Figure 2: Dinning Philosopher Running at Defferent Point of Time

Figure 2 contains four figures that shows this implementation running non-stop for four minutes. These sample output of the running program suggests that this implementation is working and without deadlocks. The main reason that this implementation is free of deadlocks is that it does not use locks at all and also this implementation uses STM which allows the threads running without the knowledge of the global environment. Giving two forks, each thread simple keep trying to acquire them at the same time until success. In this way, deadlocks can be avoided as resources will be acquired only until they are available. In contrast, if a thread acquired partially available and wait until another resource is available then deadlocks may easily occur.

There are other solutions like resource hierarchy solution and arbitrator solution available for this problem. The resource hierarchy solution follows conventions that all forks are acquired in order and only one philosopher can pick up the fork in the highest order. This solution requires that all resources are known to all the philosophers. But this is often hard to achieve in real-world use cases which makes this solution impractical. As for the arbitrator solution, resource acquisition is controlled by an arbitrator instead of philosophers. While philosophers can put down forks at any time, the permission of picking up forks is controlled by the arbitrator and the arbitrator will grant the permission to only one philosopher at a time. One obvious issue for this solution is the permission control. All philosophers have to wait for permission from the arbitrator even if there are forks available. This issue would often cause efficiency problems.

Comparing to those two solutions, STM solution is free of deadlocks and also the code is simple to write and expressive. But the STM solution is suffering from time penalty for committing atomic transactions. Also, one limitation of STM needs to be considered is that all the operations that can be performed by STM must be revertible. Despite its disadvantages and limits, STM is justified by its benefits.

2 Task II.2

Before the explanation of this implementation, the capabilities of this calculator will be introduced. The features supported by this calculator are:

- Handle at least 10-digit integers
- Support addition, subtraction, multiplication, and division
- Allow the sign to be changed (+/-)
- Allow the calculator to be reset (C) as well as clearing the last entry (CE)
- Support calculations with decimal fractions (decimal point)
- Support standard precedence rules among the arithmetic operations along with parentheses for grouping
- Have a clearly structured implementation making use of functional reactive programming

```haskell
main :: IO ()
main = do
    startGUI defaultConfig
        { jsPort       = Just 8023
        , jsStatic     = Just "static"
        } setup
```

Figure 3: Calculator Main Function

Figure 3 shows the main function of the calculator. This function is adopted from the examples provided by the threepenny library.

```haskell
setup :: Window -> UI ()
setup window = do
    -- Sets the window title
    return window # set UI.title "Calculator -- Threepenny"
    UI.addStyleSheet window "bootstrap.min.css"
    UI.addStyleSheet window "creative.css"

    but123 <- mapM mkNumButton (convert [1..3])
    but456 <- mapM mkNumButton (convert [4..6])
    but789 <- mapM mkNumButton (convert [7..9])
    reset <- mkNumButton "C"
    clearEntry <- mkNumButton "CE"
    parens <- mapM mkNumButton ["(", ")"]
    butZeroDot <- mapM mkNumButton ["0", "."]
    butCSign <- mkNumButton "+/-"
    butOps <- mapM mkNumButton ["/", "*", "-", "+"]
    butEqual <- mkNumButton "="

    let event123 = zipWith digitAction but123 (convert [1..3])
    let event456 = zipWith digitAction but456 (convert [4..6])
    let event789 = zipWith digitAction but789 (convert [7..9])
    let eventParens = zipWith digitAction parens ["(", ")"]
    let eventZeroDot = zipWith digitAction butZeroDot ["0", "."]
    let eventOps = zipWith digitAction butOps ["/", "*", "-", "+"]
    let eventClear = clearLastEntry clearEntry
    let eventReset = resetAction reset
    let eventEqual = equalAction butEqual
    let eventCSign = changeSign butCSign

    let allEvents = event123 ++ event456 ++ event789 ++
                      eventParens ++ eventZeroDot ++ eventOps ++
                    [eventReset] ++ [eventClear] ++
                    [eventEqual] ++ [eventCSign]

    calc <- accumB "" $ foldl1 (unionWith const) allEvents
```

(a) Part I

```haskell
    row1 <- UI.div #. "row" #+ ([element reset] ++
                                [element clearEntry] ++
                                map element parens)
    row2 <- UI.div #. "row" #+ (map element but789)
    row3 <- UI.div #. "row" #+ (map element but456)
    row4 <- UI.div #. "row" #+ (map element but123)
    row5 <- UI.div #. "row" #+ (element butCSign : map element butZeroDot)
    opRow <- UI.div #. "col-md-3 operationSide" #+ (map element butOps ++ [element butEqual])

    numberPad <- UI.div #. "row numberPad" #+
                 ([UI.div #. "col-md-9" #+
                   map element [row1, row2, row3, row4, row5],
                   element opRow])


    displayBox <- UI.div #. "row displayBox" #+(
      [UI.label #. "displayText" # set UI.id_ "display" # sink UI.text calc])


    calcDiv <- UI.div #. "col-md-4 col-md-offset-4 calculator" #
               set UI.align "center" #+
               (map element [displayBox, numberPad])

    getBody window #+ [UI.div #. "container" #+ [element calcDiv] ]
    return ()
```

(b) Part II

Figure 4: Calculator Setup Function

Figure 4 shows the implementation of setup function. The setup function is where logic and UI are defined. The first block of code in the setup function set the title of the window and imported two necessary CSS files for the layout. The first CSS file is the open source stylesheet called bootstrap that was developed by twitter. The second css [1] file was obtained from GitHub to make the reasonable user interface. The next block contains the code of how the UI of all the buttons is created. The mkNumButton function is used for button creation. In general, buttons that same event function can be applied to can be created. However, in this case, buttons need to be created separately for layout arranging. For buttons that created together, mapM is used to apply mkNumButton function to all buttons with different id and text while a single button creation is using mkNumButton function directly.

The on click events need to be coupled with buttons after all the buttons are created. This implementation using the parser for arithmetic expression recognition and evaluation. Therefore, all digit buttons, parentheses buttons, dot button and operation buttons(/, *, -, +) behave identically which is append text on the button to the existing input text. For all the button mentioned above, the digitAction function is used to handle the click event. The remaining buttons which are ce, reset, change sign and equal have different behaviours. Hence, each of them has a corresponding function defined to model their actions. Then, all buttons coupled with events are put in a list call allEvents. Finally, all the events are unified toghther with accumB function applied to. Until this point, all the core logics are embodied with buttons. The idea behind this embodiment is called functional reactive programming.

The next part that is shown in figure 4 (b) is mostly the definition of UI components and their layout. The part begin with how buttons are grouped into different rows and then how numberPad of the calculator is formed by combining row. One of the most important UI component, displayBox, is defined here. All the result of button events and the result of evaluation of arithmetic expression will be displayed here. Therefore, all the logic parts are combined and applied here using sink function. Together with displayBox and numberPad, the definition of this calculator is completed. And finally, the displayBox and numberPad are displayed to the window.

[1] https://github.com/xxczaki/calculator.js/blob/5d8ec0901e3503a6480d086df0d8e55c39908cdc/css/creative.css

```haskell
digitAction :: Element -> String -> Event (String -> String)
digitAction button str = (\s -> addChar s str) <$ UI.click button

addChar :: String -> String -> String
addChar s1 s2 = s1 ++ s2

changeSign ::Element -> Event (String -> String)
changeSign button = (\s -> show (csign s)) <$ UI.click button
  where
    csign str = (-1) * read (P.eval str):: Double

clearLastEntry :: Element -> Event (String -> String)
clearLastEntry button = init <$ UI.click button

resetAction :: Element -> Event (String -> String)
resetAction button = (\_ -> "") <$ UI.click button

equalAction :: Element -> Event (String -> String)
equalAction button = P.eval <$ UI.click button

mkNumButton :: String -> UI Element
mkNumButton str = UI.button #. "btn btn-calc hvr-radial-out" #
  set UI.id_ str # set UI.text str

convert :: [Int] -> [String]
convert = (>>= return.show)
```

Figure 5: Calculator Functions

Figure 5 shows functions related to button creation and button event mentioned earlier. The digitAction function takes a button and a string that is the button's text as input and return a function wrapped in Event as output. The function inside event takes the existing text as input and concatenate with the button's text and return the combined text. This is done through a dummy function called addChar.

The changeSign function defined event related to change sign button. This function takes a button as input and will first calling the parser to evaluate the existing text(arithmetic expression) then change the sign of the evaluated result.

Since all the inputs and the results are just strings. Therefore, the clear last entry can be implemented by simply calling the init function to remove the last bit. This is how the clearLastEntry function is defined.

The reset event is even simpler than the clearLastEntry event. All the need to be done is discard everything by setting it to empty string.

The final button event is the equalAction function. This function is simply passe the current input text to the parser for evaluation and returns the evaluated result.

The mkNumButton function is used to create buttons. This function set the button id and display text according to the input string, and also some predefined class is set for layout purpose.

Finally, a dummy function is defined to convert a list of integers to a list of corresponding strings.

```haskell
module Parser (eval) where
import Data.Char
import Control.Applicative

-- tree means what has parsed and string means remaining string
-- it will return a list to indicates the result of parsing
-- singleton list means success and empty means failure

-- parser with a dummy constructor
newtype Parser a = P (String -> [(a, String)])

-- apply constructor
parse :: Parser a -> String -> [(a,String)]
parse (P p) str = p str

-- parse string result first item and rest as result
-- or empty list as fail
item :: Parser Char
item = P(\str -> case str of
           []     -> []
           (x:xs) -> [(x,xs)])
-- make Parser functor, applicative and monad to combine several together
-- fmap applies function to the result of a parser
instance Functor Parser where
  -- fmap :: (a -> b) -> Parser a -> Parser b
  fmap g p = P (\str -> case parse p str of
                  [] -> []
                  [(a, rest)] -> [(g a, rest)])


instance Applicative Parser where
  -- pure :: a -> Parser a
  pure a = P (\str -> [(a, str)])

  -- <*> :: Parser (a -> b) -> Parser a-> Parser b
  pa <*> pb = P (\str -> case parse pa str of
                   [] -> []
                   [(a, rest)] -> parse (fmap a pb) rest)

instance Monad Parser where
  -- >>= :: Parser a -> (a -> Parser b) -> Parser b
  p >>= f = P (\str -> case parse p str of
                 [] -> []
                 [(a, rest)] -> parse (f a) rest)
```

(a) Parser Types

```haskell
-- expr ::= term + expr | term - expr | term
-- term ::= factor * term | factor / term | factor
-- factor ::= ( expr ) | double
-- double ::= ... -1.0 | 0.0 | 1.0 | 2.0 ...

expr :: Parser Double
expr = plus <|> minus <|> term

plus :: Parser Double
plus = do t <- term
          do symbol "+"
             e <- expr
             return (t + e)


minus :: Parser Double
minus = do t <- term
           do symbol "-"
              e <- expr
              return (t - e)


term :: Parser Double
term = times <|> divide <|> factor

times :: Parser Double
times = do f <- factor
           do symbol "*"
              t <- term
              return (f * t)


divide :: Parser Double
divide = do f <- factor
            do symbol "/"
               t <- term
               return (f / t)


factor :: Parser Double
factor = do symbol "("
            e <- expr
            symbol ")"
            return e
         <|> double
```

(b) Parser Grammer

Figure 6: Parser

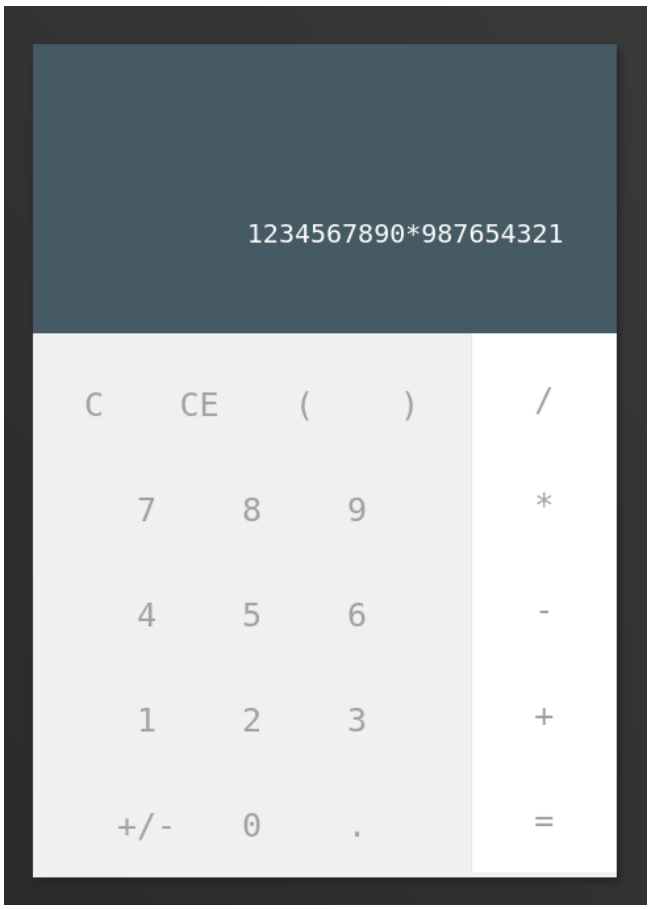
The calculator is implemented by using a parser to evaluate the arithmetic expressions and return the results. Figure 6 shows part of the parser type definition and how the arithmetic expressions are defined in context-free grammar and then converted to Haskell code.

```haskell
--parser that ignore spaces around integers
integer :: Parser Double
integer = token int

-- parser for integers based on nat parser
int :: Parser Double
int = nat <|>
      do char '-'
         n <- nat
         return ((-1) *  n)

-- parser for natural numbers
nat :: Parser Double
nat = do xs <- some digit
         return (read xs)


double :: Parser Double
double = double'
         <|> do char '-'
                d <- double'
                return (-d)
         <|> integer


double' :: Parser Double
double' = do n <- some digit
             char '.'
             decimal <- some digit
             return (read $ n ++ '.' : decimal)
```

Figure 7: Parse Doubles

This parser implementation is inspired by Graham Hutton in his book called Programming in Haskell. Most of this parser is identical to Hutton's implementation which has a thorough introduction in his book. But how to parse floating number is not mentioned. Figure 7 contains a novel implementation of how to parse floating numbers from string. And this part is essential to ensure the functionality of the calculator.

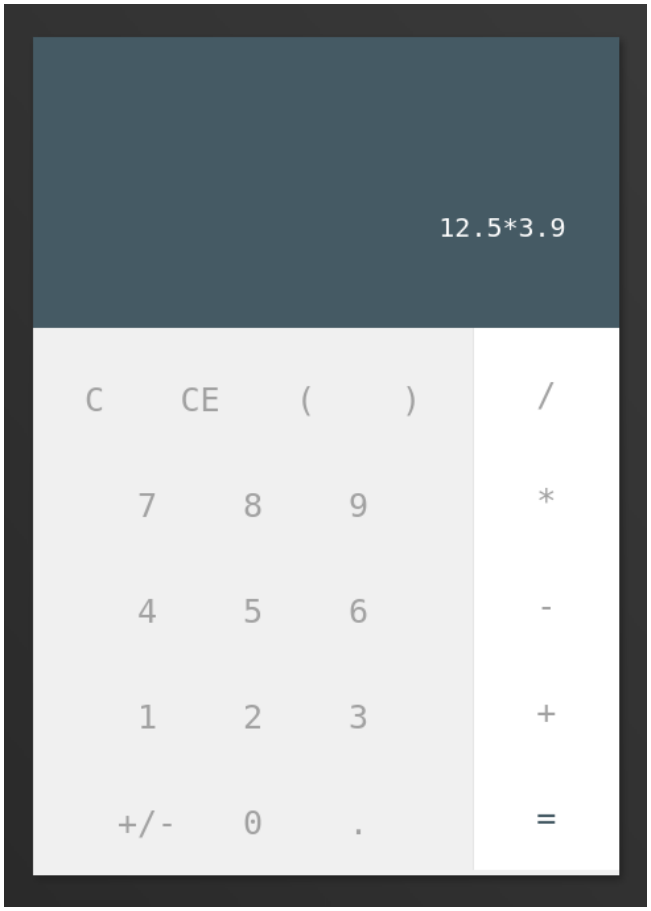
Finally, the following figures are screenshots that show the calculator in action.



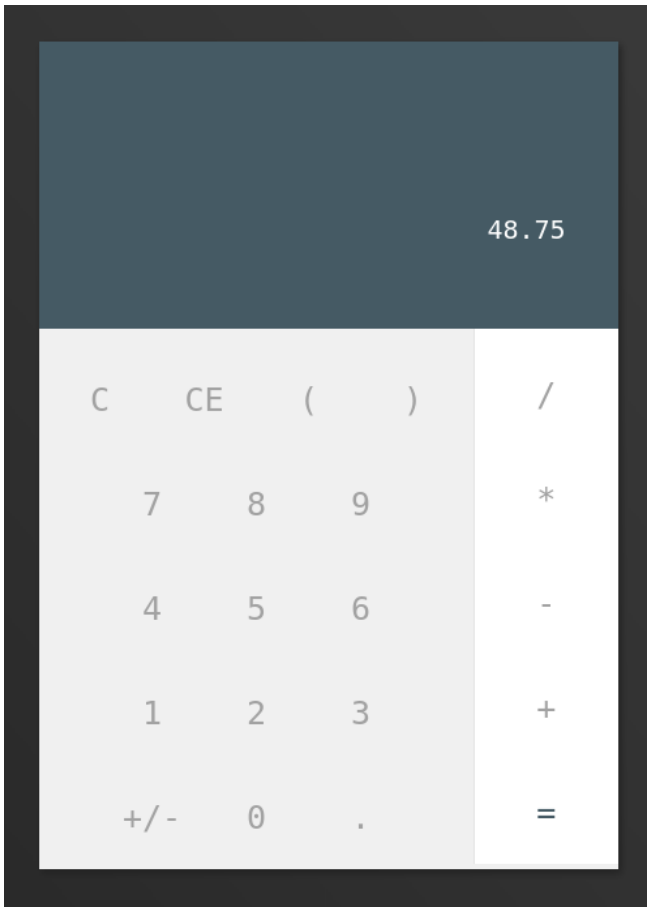
(a) Integer Multiplication

(b) Result

Figure 8: Integer Multiplication



(a) Decimal Fractions Multiplication



(b) Result

Figure 9: Decimal Fractions Multiplication

(a) Precedence Rules

(b) Result

Figure 10: Precedence Rules

(a) Paretheses Grouping

(b) Result

Figure 11: Paretheses Grouping