G54RFP Coursework Part II

Jack Ellis
psyje5@nottingham.ac.uk
4262333

Contents

1 Task II.1 - Dining Philosophers **1**
1.1 This approach versus the *Resource Hierarchy Solution* 4
1.2 This approach versus the *Arbitrator Solution* 4

2 Task II.2 - Threepenny Calculator **5**
2.1 Parsing a String to a Maybe Float 5
2.2 Applying this to Threepenny GUI 9
2.3 The Calculator In Action 11

1 Task II.1 - Dining Philosophers

```
type Spoon = TMVar Int
type Philosopher = String
```

For clarity, I firstdefined a pair of new types: `Spoon` and `Philosopher`, which map to `TMVar Int` and `String` respectively. In my implementation the Philosophers are eating with spoons, in order to avoid confusion between forks (the cutlery) and forks (the making of new threads). I then defined 3 functions to initialise, take, and return spoons.

```
newSpoon :: Int -> IO Spoon
newSpoon n = newTMVarIO n

getSpoon :: Spoon -> STM Int
getSpoon f = takeTMVar f
```

```
retSpoon :: Spoon -> Int -> STM()
retSpoon f i = putTMVar f i
```

These are essentially simple maps of the STM Monad's functions to relate them to the Spoon type.

```
philNames :: [Philosopher]
philNames = ["Graham Hutton",
             "Steve Bagley",
             "Henrik Nilsson",
             "Thorsten Altenkirch",
             "John Garibaldi"]
```

I elected to use lecturers in the School of Computer Science as my philosopher names.

```
runPhil :: Philosopher -> (Spoon, Spoon) -> IO()
runPhil p (l, r)
  = do putStrLn (p ++ " is hungry")
       (ln, rn) <- atomically $ do leftNum <- getSpoon l
                                   rightNum <- getSpoon r
                                   return (leftNum, rightNum)
       putStrLn (p ++ " has both spoons and is eating")
       delay <- randomRIO (1,maxDelay)
       threadDelay (delay*1000000)
       putStrLn (p ++ " is done eating and is thinking again")
       atomically $ do retSpoon l ln
                       retSpoon r rn
       delay <- randomRIO (1,maxDelay)
       threadDelay (delay*1000000)
       runPhil p (l, r)
```

Now for "running a philosopher". This is the infinite process by which they become hungry, get both spoons, eat, return both spoons, think, and become hungry again. This requires both spoons to be available, consequently we try to get both the left and right spoons `atomically`, ensuring that the process will halt until both are available. The thread is then delayed by a random amount between 1 and `maxDelay` (currently 5) seconds while the philosopher eats, and upon that being completed the spoons are `atomically` returned. Another delay of between 1 and `maxDelay` seconds, and the process runs again with the same arguments.

```
makePairs :: [a] -> [(a,a)]
makePairs (x:[])     = []
makePairs (x:y:[])  = (x,y):[]
makePairs (x:y:xs)  = (x,y):(makePairs (y:xs))
```

Before `main` is declared we need a function to make pairs of `Spoon`s, for passing into the `runPhil` function. This is relatively trivial.

```
main :: IO()
main
  = do spoons <- mapM newSpoon [1..length philNames]
       (putStrLn . show) philNames
       let namedPhils = map runPhil philNames
           spoonPairs = makePairs ((last spoons):spoons)
           philsWithSpoons = zipWith ($) namedPhils spoonPairs
       putStrLn "Press enter to stop"
       mapM_ forkIO philsWithSpoons
       getLine
       return ()
```

Finally, `main`, where everything is put together. First we generate a list of spoons equal to the length of the list of philosophers (whose names we print to the screen). Then we declare 3 variables:

- `namedPhils`, which is a list of partially applied `runPhil`s waiting for the `(l,r)` argument.

- `spoonPairs`, which is a list of pairs of `Spoon`s generated from the list of `Spoon`s generated earlier.

- `philsWithSpoons` is then the result of zipping `namedPhils` with `spoonPairs`, the list of functions to be forked.

Finally `mapM_` is used to fork all of the `philsWithSpoons` processes, taking only their side-effects (namely the printing). The `getLine` at the end is used to stop the function, and it `return`s empty.

In the `Philosophers` folder containing the source code is a file `output.txt` containing 65 lines of program output; the following is the first 15 lines from that file (not including the list of philosophers and ”Press enter to stop” message).

```
Graham Hutton is hungry
Graham Hutton has both spoons and is eating
```

```
Steve Bagley is hungry
Henrik Nilsson is hungry
Henrik Nilsson has both spoons and is eating
Thorsten Altenkirch is hungry
John Garibaldi is hungry
Henrik Nilsson is done eating and is thinking again
Thorsten Altenkirch has both spoons and is eating
Henrik Nilsson is hungry
Graham Hutton is done eating and is thinking again
Steve Bagley has both spoons and is eating
Thorsten Altenkirch is done eating and is thinking again
John Garibaldi has both spoons and is eating
John Garibaldi is done eating and is thinking again
```

1.1 This approach versus the *Resource Hierarchy Solution*

In the *Resource Hierarchy Solution*, the resources (spoons) are given an order of priority, and of the two required, each ”unit of work” (philosopher) will only attempt to access the lower-numbered one first. The main issue with this approach is that if the first four philosophers are to pick up their lower spoon, the fifth cannot, and a deadlock occurs. The STM approach avoids this; by not enforcing which spoon is picked up first and only requiring that both spoons are picked up at some point before eating can begin it ensures that no such deadlock can occur.

1.2 This approach versus the *Arbitrator Solution*

In the *Arbitrator Solution*, an additional entity is introduced: a waiter who must be asked before any philosopher can pick up a spoon, and who gives permission to one philosopher at a time until that philosopher has picked up both spoons. The main issue with this approach is the introduction of an additional entity (and the increased complexity associated with that), and the fact that it can result in reduced parallelism; if one philosopher is eating and a neighbour requests a spoon, no other philosophers can even request their own spoons.

2 Task II.2 - Threepenny Calculator

2.1 Parsing a String to a Maybe Float

Firstly I define a couple of data types, `Calc`, and `Op`. These will form the basis of the overall calculator and the shunting-yard algorithm, respectively.

```
data Calc = Calc {working :: String,
                  ans     :: Float,
                  mem     :: Float}
```

`working` refers to the working space; the string denoting the equation that, upon the user hitting '=', will be evaluated. I elected to model my calculator on a scientific model because I find them more intuitive to use than the traditional kind, as well as the fact that it is essentially an example of lazy evaluation; computation only occurs when the answer is demanded, a la Haskell itself. `ans` refers to the answer to the previous equation, and is the space where upon evaluation the result of the equation described in `working` will be stored. `mem` is the calculator's memory; slightly unnecessary here given the answer recall function afforded by `ans` however any Float value can be stored here.

```
data Op = Op {symbol        :: String,
              precedence    :: Int,
```

Here the `symbol` element is the symbol of the operator, used for lookup purposes. The `precedence` is the precedence of the operator; this follows BODMAS rules and the values are arbitrary. `associativity` is used for Shunting-Yard: one of the things that the algorithm checks is the associativity of the operator at the top of the stack, so that gets stored here.

```
errorMsg :: String
errorMsg = "Error_invalid_equation"

emptyCalc :: Calc
emptyCalc = Calc {working = "",
                  ans = 0.0,
                  mem = 0.0}
```

These two are used in the Threepenny section; `errorMsg` gives a consistent message when something errors out, and is used for equality, and `emptyCalc` is used on initialisation and when a user hits the 'C' button.

```haskell
ops :: [Op]
ops = [Op {symbol = "(", precedence = 6, associativity = "u"},
       Op {symbol = ")", precedence = 6, associativity = "u"},
       Op {symbol = "^", precedence = 5, associativity = "r"},
       Op {symbol = "/", precedence = 4, associativity = "l"},
       Op {symbol = "*", precedence = 3, associativity = "l"},
       Op {symbol = "+", precedence = 2, associativity = "l"},
       Op {symbol = "-", precedence = 1, associativity = "l"}]

fns :: [String]
fns = ["ln", "sin", "cos", "tan", "sqrt"]

specials :: [(String, Float)]
specials = [("pi", pi), ("e", exp 1)]
```

Here the lists of operators, functions, and special values are defined; these are defined such that the Shunting-Yard algorithm can detect whether or not a token it encounters is an operator (and if it is the precedence and associativity associated with it), a function, or a special value (currently only pi).

```haskell
isOp :: String -> Bool
isOp x = (elem x . map symbol) ops

getOp :: String -> Op
getOp op = (head . filter ((==op) . symbol)) ops

getOpPrec :: String -> Int
getOpPrec = precedence . getOp

getOpAss :: String -> String
getOpAss = associativity . getOp

isFn :: String -> Bool
isFn f = elem f fns

specialConvert :: String -> String
specialConvert s = (show . snd . head . filter ((==s) . fst)) specials
```

Now some helper functions to check whether or not a token is an operator, and get an operator's precedence or associativity. Also a function to check whether or not a token is a function, and whether or not one is a special value and convert that token to its actual value.

```
parseEqn :: String -> [String]
parseEqn s = parseEqn' (words s) ([],[])

parseEqn' :: [String] -> ([String], [String]) -> [String]
parseEqn' [] (output, stack) = output ++ stack
parseEqn' (s:ss) (output, stack)
  | isOp s
    = parseEqn' ss $ parseEqn'' s output stack
  | isFn s
    = parseEqn' ss (output, s:stack)
  | elem s $ map fst specials
    = parseEqn' ss (output ++ [specialConvert s], stack)
  | otherwise
    = parseEqn' ss (output ++ [s], stack)

parseEqn'' :: String -> [String] -> [String] -> ([String], [String])
parseEqn'' op out [] = (out, [op])
parseEqn'' ")" out (s:ss) = if s == "("
                            then (out, ss)
                            else parseEqn'' ")" (out++[s]) ss
parseEqn'' "(" out stack@(s:ss) = (out, "(":stack)
parseEqn'' op out stack@(s:ss)
  = if and [s/="(", isFn s] ||
       and [s/="(", getOpPrec s > getOpPrec op, getOpAss s == "l"]
    then parseEqn'' op (out ++ [s]) ss
    else (out, op:stack)
```

Now we define a series of functions that take a string and convert it into Reverse Polish Notation (RPN). `parseEqn` is effectively a setup function which takes the string, separates it by the delimiter (in this case spaces), and passes that and an empty tuple through to `parseEqn'`. In terms of `parseEqn'`, if it reaches the end of the list of strings it will return the output plus the operator stack. If it has tokens to evaluate, it has 4 options: if that token is an operator it will go to another helper function to determine where in the `(output, stack)` tuple it fits. This will be discussed later. If the token is a function it is cons'ed to the stack, and if it is a value (numeric or special) is is appended to the output. `parseEqn''` adds tokens to the stack in accordance with Shunting-Yard rules, which have been minimised somewhat.

```
safeEvalRPN :: [String] -> Maybe Float
safeEvalRPN = head . foldl foldingFunction []
              where foldingFunction (x:y:ys) "^"
                      = (safeEval2 (**) y x):ys
                    foldingFunction (x:y:ys) "/"
                      = (safeEval2 (/) y x):ys
                    foldingFunction (x:y:ys) "*"
                      = (safeEval2 (*) y x):ys
                    foldingFunction (x:y:ys) "+"
                      = (safeEval2 (+) y x):ys
                    foldingFunction (x:y:ys) "-"
                      = (safeEval2 (-) y x):ys
                    foldingFunction (x:xs) "ln"
                      = (safeEval1 log x):xs
                    foldingFunction (x:xs) "sin"
                      = (safeEval1 sin x):xs
                    foldingFunction (x:xs) "cos"
                      = (safeEval1 cos x):xs
                    foldingFunction (x:xs) "tan"
                      = (safeEval1 tan x):xs
                    foldingFunction (x:xs) "sqrt"
                      = (safeEval1 sqrt x):xs
                    foldingFunction xs numberString
                      = (safeRead numberString):xs

safeEval2 :: Num a => (a -> a -> a) -> Maybe a -> Maybe a -> Maybe a
safeEval2 f x y = do x' <- x
                     y' <- y
                     return $ f x' y'

safeEval1 :: Num a => (a -> a) -> Maybe a -> Maybe a
safeEval1 f x = x >>= \x' -> return $ f x'

safeRead :: String -> Maybe Float
safeRead s
  = safeRead' s s
    where safeRead' s []       = Just (read s :: Float)
          safeRead' s ('.':[]) = Nothing
          safeRead' s (s':ss') = if s' == '.' || isDigit s'
```

```
                                   then safeRead' s ss'
                                   else Nothing
```

Finally for the ”evaluating a string” section of the program, a safe evaluator of Reverse Polish Notation. `safeEvalRPN` does the overall evaluation of the RPN list of tokens to a Maybe Float. `safeEval1` and `2` are the safe evaluators for unary and binary functions respectively, and `safeRead` is a safe reader for `String`s to `Float`s.

```
safeDo :: String -> Maybe Float
safeDo = safeEvalRPN . parseEqn
```

Finally `safeDo` composites `safeEvalRPN` and `parseEqn`.

2.2 Applying this to Threepenny GUI

```
btns :: [(UI Element, Element -> Event (Calc -> Calc))]
btns = [(newButton "ln",    makeEvent (calcUpdateWorking " ln ( ")),
        (newButton "sin",   makeEvent (calcUpdateWorking " sin ( ")),
        (newButton "cos",   makeEvent (calcUpdateWorking " cos ( ")),
        (newButton "tan",   makeEvent (calcUpdateWorking " tan ( ")),
```

The main UI component here is `btns`, which is a list of tuples containing `UI Element`s and functions which take an `Element` and return an `Event`. This will be used for showing the buttons, as well as binding them to the `Calc -> Calc` functions. The structure of the list is representative of the layout of the page, and `(UI.br, id)` tuples are used for line breaks.

```
makeEvent c e = c <$ UI.click e

calcNeg :: Calc -> Calc
calcNeg c = c {ans = ((0-) . ans) c}

memAdd :: Calc -> Calc
memAdd c = c {mem = ans c}

memGet :: Calc -> Calc
memGet c = c {working = working c ++ " " ++ (show . mem) c}

memClear :: Calc -> Calc
memClear c = c {mem = 0.0}
```

```
calcAddAns :: Calc -> Calc
calcAddAns c = calcUpdateWorking (" " ++ (show . ans) c ++ " ") c
```

All of the above could have been incoroporated into a `where` clause in the `btns` declaration, but for clarity's sake I thought it best to define them separately, with their type signatures. `newButton` and `makeEvent` are functions that take a string and return a button with that string as a label, and make an event out of some variable given to it respectively. They are used in `btns` for easy generation of buttons and associated events. `calcNeg` is used to make the value stored in

```
calcUpdateWorking :: String -> Calc -> Calc
calcUpdateWorking s c
  | wc == "" = if elem (filter (/=' ') s) ((map symbol . drop 2) ops)
                 then (calcUpdateWorking s . calcUpdateWorking ((show . ans) c))
                 else c {working = s}
  | wc == errorMsg = calcUpdateWorking s $ c {working = ""}
  | otherwise = c {working = wc ++ s}
  where wc = working c
        ac = ans c
        mc = mem c
```

`calcUpdateWorking` updates the working space of the calculator; if the value being added is an operator and the space is currently empty it will perform that operator on the current answer value. If the working space is occupied by an error message, it will be overwritten and ignored. Otherwise the new token will be appended to the working space.

```
calcEval :: Calc -> Calc
calcEval c =
  case (safeDo . working) c
    of Nothing -> c {working = errorMsg}
       Just n  -> c {working = "", ans = n}
```

`calcEval` takes the `Calc` and, using the string evaluator defined above, updates it based on the contents of the working space. If the result of evaluating it is `Nothing` it will print an error message to the screen, otherwise it will have the correct value in the answer space and allow the user to input more.

```
setup :: Window -> UI ()
setup window =
  do return window # set UI.title "Calculator"
```

```
    buttons <- (sequence . map fst) btns
    let btnFns = unionList const (zipWith ($) (fmap snd btns) buttons)
    calc <- accumB emptyCalc btnFns
    work <- UI.label # sink UI.text (fmap (filter (/=' ') . working) calc)
    answer <- UI.label # sink UI.text (fmap (show . ans) calc)
    getBody window #+ [UI.center #+ ([element work,
                                      UI.br,
                                      element answer,
                                      UI.br]
                                      ++ (map element buttons))]
    return ()
    where unionList c (e:[]) = e
          unionList c (e:es) = unionWith c e $ unionList c es
```

`setup` sets up the FRP elements, drawing the buttons, marrying buttons with the functions they call, and drawing the workspace and answer sections.

```
main :: IO ()
main =
  do startGUI defaultConfig {jsPort = Just 8023,
                             jsStatic = Just "../wwwroot"} setup
```

Finally, `main` sets up the whole thing and starts the web server.

2.3 The Calculator In Action

In figure 1 we can see the calculator on a fresh page open. The buttons function as expected by and large, numbers and functions appending to the working space, the `Ans` key appending to it the value currently stored in the main `Calc` variable's `ans` record (displayed just below the working space. The memory buttons work as follows:

- `M+`: sets the memory to the current answer value.
- `M`: appends the current memory value to the working space.
- `MC`: clears the memory (sets it to 0).

`C` and `CE` do not affect the memory value; this is persistent until the user either closes the calculator or clears it.

Figure 2 shows the calculator in the middle of being used. Here the working space contains `sin(5)`, and the answer value is `101.0`. The value of the working space will not be passed into the answer field until the user presses the `=` button.

If the working space is empty and the user inputs an operator, that operator will take as its first argument the current answer value, i.e. if the working space was empty in figure 2 and I hit the `-` key the working space would then read `101.0-`.

Figure 3 shows the calculator before and after attempting to evaluate an invalid expression; this one has mismatched brackets, however the calculator will return this error message on any poorly-formed equation (i.e. multiple operators without numbers in between). If a user then adds to the working space the error message is automatically overwritten and calculation can continue as normal.

Figure 1: The calculator as it starts up

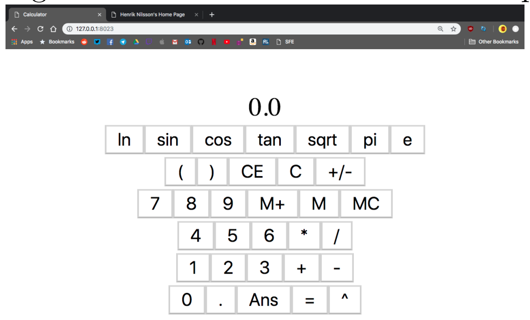


Figure 2: The calculator in use

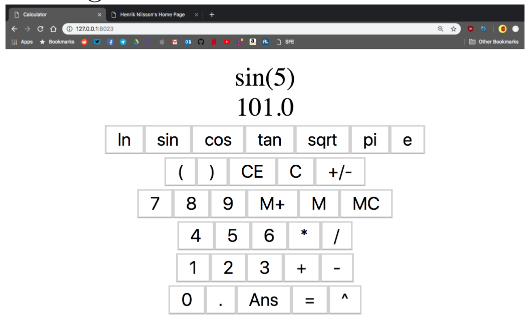
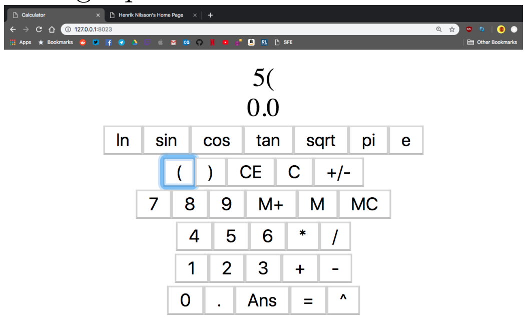
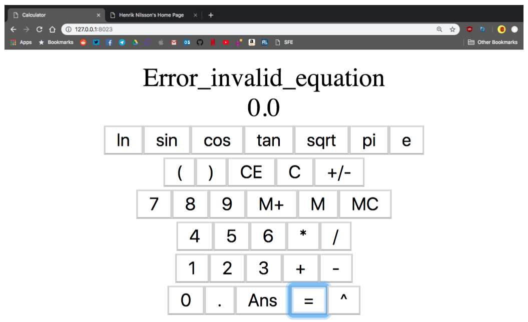


Figure 3: The calculator with an invalid expression in the working space, and the result of pressing equals



5(
0.0



Error_invalid_equation
0.0