

Haskell Group Progress Report

Members:

Del Mundo, Guiane Carlo

Lumba, Nelwyn Jairoh

Pugal, Reiven Curt

Rivera, Kurt Francis

This week, we have tried incorporating file handling in our program. Currently, it can save to and load from a csv file. Next week, we will try to incorporate design elements in our program since we do not have access to resources to make a graphical user interface.

The screenshot displays the Online Haskell Compiler IDE interface. On the left, a file explorer shows a 'New Project' with a single file 'Main.hs'. The main editor area contains the Haskell code for a payroll system. The code defines an 'Employee' data structure with fields for ID, name, position, salary, and hours worked. It includes a 'main' function that runs a menu-driven program. The menu options are: 1. Add Employee, 2. Edit Employee, 3. Delete Employee, 4. Compute Salary, 5. Show All Employees, 6. Exit. The program uses a 'do' block to handle user input and state changes. The output window on the right shows the program's execution, including the menu display and user input for adding a new employee.

```
1 import Data.List
2 import Data.Maybe
3 import System.IO
4
5 -- Data structure of Employee
6 data Employee = Employee {
7   empId      :: Int,
8   empName    :: String,
9   empPosition :: String,
10  empSalary   :: Float, -- hourly rate
11  empHours    :: Float  -- hours worked
12 } deriving (Show, Eq)
13
14 type Payroll = [Employee]
15
16 -- MAIN LOOP
17 main = do
18   putStrLn "==== Simple Payroll System ===="
19   mainMenu []
20
21 -- MAIN MENU
22 mainMenu :: Payroll -> IO ()
23 mainMenu payroll = do
24   putStrLn "Choose an option:"
25   putStrLn "1. Add Employee"
26   putStrLn "2. Edit Employee"
27   putStrLn "3. Delete Employee"
28   putStrLn "4. Compute Salary"
29   putStrLn "5. Show All Employees"
30   putStrLn "6. Exit"
31   choice <- getLine
32   putStrLn "Enter your choice: "
33   MaybeM $ do
34     case choice of
35       "1" -> do
36         newPayroll <- addEmployee payroll
37         mainMenu newPayroll
38       "2" -> do
39         newPayroll <- editEmployee payroll
40         mainMenu newPayroll
41       "3" -> do
42         newPayroll <- deleteEmployee payroll
43         mainMenu newPayroll
44       "4" -> do
45         computeSalary payroll
46         mainMenu payroll
47       "5" -> do
48         showEmployees payroll
49         mainMenu payroll
50       "6" -> do
51         putStrLn "Exiting Payroll System. Goodbye!"
52         return ()
53       _ -> do
54         putStrLn "Invalid option. Try again."
55         mainMenu payroll
```

Output:

```
[1 of 1] Compiling Main (jsoondie.hs, jsoondie.o)
Linking jsoondie ...
==== Simple Payroll System ====

Choose an option:
1. Add Employee
2. Edit Employee
3. Delete Employee
4. Compute Salary
5. Show All Employees
6. Exit
Enter your choice: 1

-- Add Employee --
Enter Employee ID: berry
Enter Employee Name: 123
Enter Position: point guard
Enter Hourly Rate: 86
Enter Hourly Worked: 1
Employee added!

Choose an option:
1. Add Employee
2. Edit Employee
3. Delete Employee
4. Compute Salary
5. Show All Employees
6. Exit
Enter your choice: 5
```

Program Code

```
1 import Data.List
2 import Data.Maybe
3 import System.IO
4
5 data Employee = Employee {
6     empId      :: Int,
7     empName    :: String,
8     empPosition :: String,
9     empRate    :: Float,
10    empHours    :: Float
11 } deriving (Show, Eq)
12
13 type Payroll = [Employee]
14 fileName :: String
15 fileName = "payroll.txt"
16
17 main :: IO ()
18 main = do
19     putStrLn "=== Simple Payroll System ==="
20     payroll <- loadEmployees
21     mainMenu payroll
22
23 mainMenu :: Payroll -> IO ()
24 mainMenu payroll = do
25     putStrLn "\nChoose an option:"
26     putStrLn "1. Add Employee"
27     putStrLn "2. Edit Employee"
28     putStrLn "3. Delete Employee"
29     putStrLn "4. Compute Salary"
30     putStrLn "5. Show All Employees"
31     putStrLn "6. Exit"
32     putStr "Enter your choice: "
33     hFlush stdout
34     choice <- getLine
35     case choice of
36         "1" -> do
37             newPayroll <- addEmployee payroll
38             saveEmployees newPayroll
39             mainMenu newPayroll
40         "2" -> do
41             newPayroll <- editEmployee payroll
42             saveEmployees newPayroll
43             mainMenu newPayroll
44         "3" -> do
45             newPayroll <- deleteEmployee payroll
46             saveEmployees newPayroll
47             mainMenu newPayroll
48         "4" -> do
```

```

49         computeSalary payroll
50         mainMenu payroll
51     "5" -> do
52         showEmployees payroll
53         mainMenu payroll
54     "6" -> putStrLn "Exiting Payroll System. Goodbye!"
55     -   -> do
56         putStrLn "Invalid option. Try again."
57         mainMenu payroll
58
59 addEmployee :: Payroll -> IO Payroll
60 addEmployee payroll = do
61     putStrLn "\n== Add Employee =="
62     putStr "Enter Employee ID: "
63     hFlush stdout
64     idStr <- getLine
65     let eid = read idStr :: Int
66     if any (\e -> empId e == eid) payroll then do
67         putStrLn "Employee ID already exists!"
68         return payroll
69     else do
70         putStr "Enter Employee Name: "
71         hFlush stdout
72         name <- getLine
73         putStr "Enter Position: "
74         hFlush stdout
75         pos <- getLine
76         putStr "Enter Hourly Rate: "
77         hFlush stdout
78         rateStr <- getLine
79         let rate = read rateStr :: Float
80         putStr "Enter Hours Worked: "
81         hFlush stdout
82         hoursStr <- getLine
83         let hours = read hoursStr :: Float
84         let newEmp = Employee eid name pos rate hours
85         putStrLn "Employee added!"
86         return (payroll ++ [newEmp])
87
88 editEmployee :: Payroll -> IO Payroll
89 editEmployee payroll = do
90     putStrLn "\n== Edit Employee =="
91     putStr "Enter Employee ID to Edit: "
92     hFlush stdout
93     idStr <- getLine
94     let eid = read idStr :: Int
95     case find (\e -> empId e == eid) payroll of
96         Nothing -> do

```

```

97         putStrLn "Employee not found!"
98         return payroll
99     Just emp -> do
100         putStrLn $ "Editing " ++ empName emp
101         putStr "Enter new Name (leave blank to keep current): "
102         hFlush stdout
103         name <- getLine
104         putStr "Enter new Position (leave blank to keep current): "
105         hFlush stdout
106         pos <- getLine
107         putStr "Enter new Hourly Rate (leave blank to keep current): "
108         hFlush stdout
109         rateStr <- getLine
110         putStr "Enter new Hours Worked (leave blank to keep current): "
111         hFlush stdout
112         hoursStr <- getLine
113         let updatedEmp = Employee
114             eid
115             (if null name then empName emp else name)
116             (if null pos then empPosition emp else pos)
117             (if null rateStr then empRate emp else read rateStr)
118             (if null hoursStr then empHours emp else read hoursStr)
119         let newPayroll = updatedEmp : filter (\e -> empId e /= eid) payroll
120         putStrLn "Employee updated!"
121         return newPayroll
122
123 deleteEmployee :: Payroll -> IO Payroll
124 deleteEmployee payroll = do
125     putStrLn "\n== Delete Employee =="
126     putStr "Enter Employee ID to Delete: "
127     hFlush stdout
128     idStr <- getLine
129     let eid = read idStr :: Int
130     if any (\e -> empId e == eid) payroll then do
131         let newPayroll = filter (\e -> empId e /= eid) payroll
132         putStrLn "Employee deleted!"
133         return newPayroll
134     else do
135         putStrLn "Employee not found!"
136         return payroll
137
138 computeSalary :: Payroll -> IO ()
139 computeSalary payroll = do
140     putStrLn "\n== Compute Salary =="
141     putStr "Enter Employee ID: "
142     hFlush stdout
143     idStr <- getLine
144     let eid = read idStr :: Int

```

```

145     case find (\e -> empId e == eid) payroll of
146     Nothing -> putStrLn "Employee not found!"
147     Just emp -> do
148         let basicSalary = empRate emp * empHours emp
149         let deductions = basicSalary * 0.10
150         let netSalary = basicSalary - deductions
151         putStrLn $ "\nEmployee: " ++ empName emp
152         putStrLn $ "Position: " ++ empPosition emp
153         putStrLn $ "Basic Salary: $" ++ show basicSalary
154         putStrLn $ "Deductions (10%): $" ++ show deductions
155         putStrLn $ "Net Salary: $" ++ show netSalary
156
157 showEmployees :: Payroll -> IO ()
158 showEmployees [] = putStrLn "\nNo employees found!"
159 showEmployees payroll = do
160     putStrLn "\n== Employee Records =="
161     mapM_ printEmployee payroll
162
163 printEmployee :: Employee -> IO ()
164 printEmployee emp = do
165     putStrLn $ "ID: " ++ show (empId emp)
166     putStrLn $ "Name: " ++ empName emp
167     putStrLn $ "Position: " ++ empPosition emp
168     putStrLn $ "Hourly Rate: $" ++ show (empRate emp)
169     putStrLn $ "Hours Worked: " ++ show (empHours emp)
170     putStrLn "-----"
171
172 saveEmployees :: Payroll -> IO ()
173 saveEmployees payroll = do
174     let empData = unlines $ map serializeEmployee payroll
175     writeFile fileName empData
176     putStrLn "Employee data saved to file!"
177
178 loadEmployees :: IO Payroll
179 loadEmployees = do
180     exists <- doesFileExist fileName
181     if not exists
182     then return []
183     else do
184         content <- readFile fileName
185         let emplyines = lines content
186         return (map deserializeEmployee emplyines)
187
188 serializeEmployee :: Employee -> String
189 serializeEmployee emp = intercalate "," [
190     show (empId emp),
191     empName emp,
192     empPosition emp,

```

```

193     show (empRate emp),
194     show (empHours emp)
195 ]
196
197 deserializeEmployee :: String -> Employee
198 deserializeEmployee str =
199     let [eid, name, pos, rate, hours] = split ',' str
200     in Employee (read eid) name pos (read rate) (read hours)
201
202 split :: Char -> String -> [String]
203 split delim str = case break (== delim) str of
204     (a, ':b) -> a : split delim b
205     (a, "") -> [a]
206

```

Inputs:

choice(String) - input used by the user to navigate the program

idStr(String) - read input for employee ID

name(String) - input for employee name

pos(String) - input for employee position in the company

rateStr(String) - input for employee's hourly rate

hoursStr(String) - input for hours worked of the employee

Note: all read inputs in Haskell is in the String datatype, the program converts the String into the appropriate datatype for each employee information

Declarations:

Employee(Data Type) – similar to class in Java, this sets up an object or a data type with the attributes empId, empName, empPosition, empRate, and empHours.

Payroll(List) – contains the list of employees in the program

main – is the entry point of the Haskell program

mainMenu – the function that contains the choices and the employee function options and control flow functionality

addEmployee – adds an employee to the payroll list

editEmployee – edits an existing employee's details

deleteEmployee – deletes an employee in the payroll list

computeSalary – computes the salary of an employee based on hours worked, hourly rate, and deductions

showEmployees – shows the employee database

saveEmployees – saves the payroll list into a text file

loadEmployees – reads the text file containing the employees and deserializes it

serializeEmployee – turns the employee data type into a single line string

deserializeEmployee – turns the single-line string from the text file and converts it into an employee datatype

split – like split in Python, this function takes an argument and splits a string into a list.

Loops and Conditions:

mainMenu – the main menu loop of the program. This loop will not terminate unless the user inputs 6, which is the exit input.

addEmployee – checks a condition if an employee with the same ID exists

editEmployee – checks a condition if the employee that the user is trying to edit exists

deleteEmployee – checks a condition if the employee that the user is trying to delete exists

showEmployees – using mapM_, the program loops through each employee data type in the list and prints it out.