مستند پروژه

طبق تعریف صورت پروژه، دو نوع دستور داریم. نوع اول دستور های Setup هستند که نوع حساب (accountType) و نوع وام (loanType) را مشخص می کند و نوع دوم دستور های Command است که بر اساس آنها کل محاسبات پروژه انجام میشود.
قدم اول در این پروژه این بود که ساختار پروژه را تعریف کنیم، در این پروژه ما از struct ها استفاده کردیم و داده های مورد نیاز را در struct ها ذخیره کردیم. داده ها را در struct، set می کنیم و بعدا داده ها را get میکنیم و به این ترتیب از تغییرات داده ها هم پشتیبانی می کنیم. کل روند پروژه همین است که با داده های جدید کار می کنیم. دو struct برای دستورات setup داریم که بالا تر اشاره شد و شامل accountType و loanType می شود، علاوه بر این دو struct، سه struct دیگر هم تعریف کردیم برای customer، account و loan. این سه تا نشانگر داده هایی هستند که در پروژه با آن ها سر و کار داریم و پس از خواندن صورت پروژه به سادگی می توان داده های اصلی این پروژه را شناسایی کرد. روابط این داده ها به این صورت تعریف می شود که هر **مشتری** یک **حساب** دارد و ممکن است چند **وام** داشته باشد.

```
;;-------------------------------------------------------------
;;-------------------- struct definition ----------------------
;;-------------------------------------------------------------

(struct AccountType (type currentAccount bankFee minDep monthly period renewable interestRate credit va
(struct LoanType (type amount blockingMoney span interest lastLoan minCredit) #:mutable #:transparent)

(struct Customer (id account loans) #:mutable #:transparent)
(struct Account (accountType balance minBalance credit timeOpened reopenTime firstBalance interestRate
(struct Loan (loanType startMonth) #:mutable #:transparent)
```

شکل ۱. struct های موجود

پس از مشخص کردن داده های اصلی تحت عنوان struct، قدم بعدی تعریف کردن دستور های global بود. یک ماه کلی(month) داریم که مثل یک شمارنده، هر ماه یک عدد به آن اضافه می شود. چند تا مشتری داریم که با customers آنها را مشخص کرده ایم و نوع حساب ها و نوع وام ها را داریم که با accountTypes و loanTypes آن ها را نشان می داده ایم و در آخر هم دستورات setup و command هستند.

```
;;-------------------- global variables ----------------------
;;-------------------------------------------------------------

;; Initialize
(define setups `())
(define commands `())

;; primary variables
(define month 0)
(define customers `())
(define loanTypes `())
(define accountTypes `())
```

شکل۲. دستور های global

در قسمت Input Processing Function، command و setup را از هم جدا کردیم. یعنی ابتدا کل متن را خواندیم و سپس بر اساس اینکه در کدام نوع از دستور ها قرار دارند جدا کردیم. به این صورت که قسمت اول که نوشته بود setup، را خواندیم و در آرایه ای ریختیم که نشان دهنده دستورات این بخش است. برای دستورات command هم، همین کار را انجام می دهیم و دستورات را در آرایه global به نام command قرار دادیم. سپس دستورات هر قسمت را به طور جداگانه اجرا میکنیم. روش اجرا این گونه است که تابعی داریم به اسم doSetupFunction که بررسی میکند که اگر دستور AccountType بود، یک account میسازد و یک accountType می سازد و اگر Loan بود یک struct برای LoanType می سازد.

```
(define (doSetupFunction multiLines)
  (cond
    [(null? multiLines) #f]
    [(equal? (getNS multiLines 0 0) "Account")
     (set! accountTypes (append accountTypes (list (AccountType
                                                    (getNS multiLines 0 2)
                                                    (string->boolean (getNS multiLines 1 1))
                                                    (string->number (getNS multiLines 2 1))
                                                    (string->number (getNS multiLines 3 1))
                                                    (string->boolean (getNS multiLines 4 1))
                                                    (string->number (getNS multiLines 5 1))
                                                    (string->boolean (getNS multiLines 6 1))
                                                    (string->number (getNS multiLines 7 1))
                                                    (string->number (getNS multiLines 8 1))
                                                    (string->boolean (getNS multiLines 9 1))
                                                    (string->number (getNS multiLines 10 1))
                                                    (string->number (getNS multiLines 11 1))
                                                    (string->boolean (getNS multiLines 12 1))
                                                    (string->boolean (getNS multiLines 13 1))
                                                    (string->number (getNS multiLines 14 1))))))]
    [(equal? (getNS multiLines 0 0) "Loan")
     (set! loanTypes (append loanTypes (list (LoanType
                                              (getNS multiLines 0 2)
                                              (string->number (getNS multiLines 1 1))
                                              (string->number (getNS multiLines 2 1))
                                              (string->number (getNS multiLines 3 1))
                                              (string->number (getNS multiLines 4 1))
                                              (string->number (getNS multiLines 5 1))
                                              (string->number (getNS multiLines 6 1))))))]
    [else #f])
  )
```

شکل ۳. تابع doSetupFunction

سپس در doCommandFunction تمامی command ها اجرا می شود و عملکرد آن بر اساس جایگیری کلمات در command ها می باشد. مثلا اگر بر اساس اینکه واژه سوم wants یا adds است، Item ها را جدا میکنیم و برای هر کدام از آنها تابع مربوطه را اجرا میکنیم.

```
(define (doCommandFunction line)
  (cond
    [(null? line) #f]
    [(equal? (getS line 2) "by")
     (nextMonth)]
    [(equal? (getS line 2) "wants")
     (let ([id (getS line 1)]
           [type (getS line 9)]
           [initBalance (getS line 16)])
       (let ([accountType (searchAccountType type accountTypes)])
         (unless (equal? accountType #f)
           (openAccount id accountType (string->number initBalance)))
         )
       )]
    [(equal? (getS line 2) "adds")
     (let ([id (getS line 1)]
           [amount (getS line 3)])
       (let ([usr (searchCustomer id customers)])
         (unless (equal? usr #f)
           (addToAccount usr (string->number amount)))
         )
       )]
    [(and(equal? (getS line 2) "requests")(equal? (getS line 3) "renewal"))
     (let ([id (getS line 1)])
       (let ([usr (searchCustomer id customers)])
         (unless (equal? usr #f)
           (requestRenewal usr))
         )
       )]
    [(equal? (getS line 2) "writes")
     (let ([id (getS line 1)]
           [amount (getS line 6)])
       (let ([usr (searchCustomer id customers)])
         (unless (equal? usr #f)
           (writeCheque usr (string->number amount))
           )
```

شکل ۴. قسمتی از تابع doCommanFunction

یک سری تابع هستند که پس از خواندن دستورات آنها را اجرا میکنیم، مثلا یک customer میخواهد حساب باز کند، پس تابع openAccount را فراخوانی میکنیم و اجرا میکنیم. یا در یک سناریو دیگر مشتری می خواهد به حسابش یک مقدار واریز کند، در این حالت addToAccount را اجرا میکنیم.

```
;; Customer 1 wants to create an account of type 1. Customer 1 wants to start with 3000000 tomans.
(define (openAccount id accountType initBalance)
  (let ([fee (AccountType-bankFee accountType)]
        [minBal (AccountType-minDep accountType)])
    (let ([newBal (- initBalance fee)])
      (when(> newBal minBal)
         (let ([newAccount (Account accountType newBal newBal 0  month month newBal (AccountType-increaseRate accountType) 0 #f)])
           (set! customers (append customers (list (Customer id newAccount `()))))
         )
      )
    )
  )

```

شکل ۵. تابع openAccount برای افتتاح حساب مشتری

یک سری توابع دیگر هستند که باید ماهانه اجرا شوند. به این صورت که هر وقت که متغیر month افزایش می یابد، باید اجرا شوند. این عملیات در تابع nextMonth انجام می شود و عملیات مربوط به آنها در این تابع قرار دارد.

```
270  ;;-----------------------------------------------------------------
271  ;;-------------------- command functions -------------------------
272  ;;-----------------------------------------------------------------
273
274  ;; Time goes by.
275  (define (nextMonth)
276    (begin(set! month (+ 1 month))
277          (spanForIncrease customers)
278          (setMinBalance customers)
279          (monthlyProfit customers)
280          (yearlyProfit customers)
281          (loanAdds customers))
282    )
283
```

شکل ۶. تابع nextMonth و عملیاتی که داخل آن صورت میگیرد.

در ابتدای کد یک کتابخانه آورده شده است که برای خروجی گرفتن استفاده کرده ایم از آن و بدین ترتیب خروجی راحت تری بر روی فایل می توانیم داشته باشیم. با این کتابخانه به راحتی میتوانیم با استفاده از (write-file "output.txt" (outputFunction)) خروجی مد نظر را بر روی فایل مشاهده کنیم.

```
#lang racket
(require 2htdp/batch-io)

::-----------------------------------------------------------------
```

شکل ۷. کتابخانه استفاده شده

```
(begin
  (process lines 0)
  (processLines setups)
  (processLines commands)
  (write-file "output.txt" (outputFunction)))
```

شکل ۸. خروجی گرفتن با استفاده از کتابخانه استفاده شده

در قسمت getting functions به جای حلقه از توابع بازگشتی استفاده شده است و با آن ها می توانیم جستجو های مورد نظرمان را انجام دهیم. مثلا برای جستجوی یک کاربر یا جستجوی AccountType از این توابع بازگشتی بهره برده ایم.

```
;; search for customer by id
;; return false if notfound
(define (searchCustomer id customers)
  (cond
    [(null? customers) #f]
    [(equal? id (Customer-id (car customers))) (car customers)]
    [else (searchCustomer id (cdr customers))]
    )
  )

;; search for AccountType by type
;; return false if notfound
(define (searchAccountType type accountTypes)
  (cond
    [(null? accountTypes) #f]
    [(equal? type (AccountType-type (car accountTypes))) (car accountTypes)]
    [else (searchAccountType type (cdr accountTypes))]
    )
  )
```

شکل ۹. استفاده از توابع بازگشتی برای جستجو

برای ایجاد یک وام تابع createLoan را پیاده سازی کرده ایم. این تابع دو پارامتر میگیرد شامل نوع وام و شناسه مشتری (id). سپس مشتری را با تابع searchCustomer پیدا میکنیم و پس از آن نوع وام را نیز توسط searchLoanType پیدا میکنیم و اگر خروجی این دو تابع false نبود یعنی مشتری وجود داشته و نوع وام نیز وجود داشته پس به سایر عملیات این تابع می پردازیم. هر مشتری یک حساب دارد. حساب مشتری را پیدا میکنیم و با دستور when سه شرط موجود را بررسی میکنیم. این سه شرط تضمین کننده این است که موجودی حساب مشتری برابر کمینه لازم برای آن نوع وام است و balance حساب وی نیز حداقل اندازه blockingMoney است و از وام قبلی ای که مشتری دریافت کرده است نیز مدت زمانی گذشته است. اگر این سه شرط همگی موجود بودند یک newLoan می سازیم و آن را به مشتری assign میکنیم. کد این تابع در صفحه بعد آورده شده است.

```
;;Customer 1 requests a loan of type 1.
(define (createLoan id loanType)
  (let ([customer (searchCustomer id customers)]
        [loanType (searchLoanType loanType loanTypes)])
    (unless (or (equal? loanType #f) (equal? customer #f))
      (displayln (Account-credit (Customer-account customer)))
      (let ([account (Customer-account customer)])
        (when
            (and
             (>= (Account-credit account) (LoanType-minCredit loanType))
             (>= (Account-balance account) (LoanType-blockingMoney loanType))
             (>= (- month (getLastLoan (Customer-loans customer))) (LoanType-lastLoan loanType)))
          (displayln "Loan got!")
          (let ([newLoan (Loan loanType (+ month 1) #f)])
            (set-Customer-loans! customer (append (Customer-loans customer) (list newLoan))))
          ))
      )
    )
  )
```

شکل ۱۰. تابع createLoan و واگذاری وام به یک مشتری

برای محاسبه سود ماهانه تابع monthlyProfit را پیاده سازی کردیم. در این تابع اینگونه عمل میکنیم که اگر customer ها null نبود یعنی حداقل حداقل یک مشتری داشته باشیم. حساب مشتری را پیدا میکنیم و از طریق حساب مشتری، نوع حساب و balance حسابش را بدست می آوریم و سپس ۳ شرط را بررسی میکنیم. این شروط شامل جاری بودن حساب و سود تعلق گرفتن به آن است و اینکه سود ماهانه داشته باشد و هر ماه به حساب سود تعلق بگیرد و اینکه از حساب وی منقضی نشده باشد. اگر این ۳ شرط برقرار بودند محاسبات مربوط به سود را انجام می دهیم. در این محاسبات عدد ۱۲۰۰ را مشاهده میکنیم که برای تعداد ماه های سال و درصد در نظر گرفته شده. از exact-floor هم برای گرد کردن اعداد اعشاری استفاده میکنیم. مثلا ۱۲.۳ را به ۱۲ تبدیل میکند.

```
(define (monthlyProfit customers)
  (unless (null? customers)
    (let ([account (Customer-account (car customers))])
      (let ([balance (Account-balance account)]
            [accountType (Account-accountType account)])
        (when (and (not (AccountType-currentAccount accountType))
                   (AccountType-monthly accountType)
                   (>= (+ (Account-reopenTime account) (AccountType-period accountType)) month))
          (set-Account-balance! account (exact-floor (+ balance (* balance (/ (Account-interestRate account) 1200))))))))
    (monthlyProfit (cdr customers))
    )
  )
```

شکل ۱۱. محاسبه سود ماهانه

تابع yearlyProfit هم کاملا مشابه تابع بالایی یا همان monthlyProfit عمل میکند با این تفاوت که هر ۱۲ ماه که میگذرد یک بار این عملیات را انجام میدهیم. تفاوت دیگر این است که اینجا دیگر تقسیم بر ۱۲۰۰ نمیکنیم چون دیگر یک سال است و فقط تقسیم بر ۱۰۰ میکنیم چون interestRate برای یک سال محاسبه می شود.

```
(define (yearlyProfit customers)
  (unless (null? customers)
    (let ([account (Customer-account (car customers))])
      (let ([balance (Account-balance account)]
            [accountType (Account-accountType account)])
        (begin
          (when (equal?  0 (remainder (- month (Account-timeOpened account)) 12))
            (set-Account-credit! account (+ (Account-credit account) (AccountType-credit accountType))))
          (when (and (equal?  0 (remainder (- month (Account-reopenTime account)) 12))
                     (not (AccountType-currentAccount accountType))
                     (not (AccountType-monthly accountType))
                     (>= (+ (Account-reopenTime account) (AccountType-period accountType)) month))
            (set-Account-balance! account (+ balance (* (Account-minBalance account) (/ (Account-interestRate account) 100))))))))
    (yearlyProfit (cdr customers))
    )
  )
```

شکل ۱۲. محاسبه سود سالانه

در ابتدای کد یک تابع lines داریم که کارایی آن اینگونه است که یک فایل را میخوانیم و آنرا تبدیل میکنیم به آرایه و آن را دی lines قرار میدهیم.

```
(define lines (file->lines "sample/sample_input.txt"))
(define (process lines flag)
  (cond
    [(null? lines) #f]
    [(eqv? (string-length (car lines)) 0) (process (cdr lines) flag)]
    [(equal? "setup" (car lines)) (process (cdr lines) 0)]
    [(equal? "commands" (car lines)) (process (cdr lines) 1)]
    [(eqv? flag 0) (begin (set! setups (append setups (list (car lines)))) (process (cdr lines) 0))]
    [(eqv? flag 1) (begin (set! commands (append commands (list (car lines)))) (process (cdr lines) 1))]
    )
  )
```

شکل ۱۳. خواندن یک فایل در ابتدای برنامه

در این پروژه هنگام کد زدن، بخش عمده ای از حواس و تلاش ما متمرکز بر این موضوع بوده است که کدی که تحویل می دهیم خوانایی بالایی داشته باشد و اسم متغییر ها و توابع به خودی خود گویای functionality آنها باشد. همچنین در تمامی قسمت های کد نیز کامنت هایی قرار داده ایم تا ابهامات احتمالی برطرف شود و برای خوانندگان کد مشکلات زیادی ایجاد نشود.