Written Assignment (Unit 3)

[The student's name has been redacted]

Programming Languages (MSIT 5216-01 - AY2025-T3)

Dr. Jikitsha Joshi

February 20, 2025

Functional Programming Implementation: Overdue Fee Calculations

Functional programming has emerged as a powerful paradigm for developing robust and maintainable software systems, including library management systems. The scenario presented involves a librarian team developing a library management system using a functional programming language, specifically focusing on managing book inventory, handling user checkouts, and calculating overdue fees. This approach aligns well with the principles of functional programming, which emphasize immutability, pure functions, and declarative programming (Krishnamurthi, 2017).

1. *Deriving lambda functions to calculate overdue fees (Task 1)*

In the context of a library management system, Racket, a dialect of Lisp, provides an excellent platform for implementing functional programming concepts. Racket's support for first-class functions and lambda expressions makes it particularly suitable for creating modular and adaptable systems (Felleisen et al., 2018).

Let's begin by deriving lambda functions to calculate overdue fees based on book categories and user-specific rules:

```
#lang racket

;; Basic fee calculation
(define basic-fee
  (λ (days-overdue daily-rate)
    (* days-overdue daily-rate)))

;; Fiction books fee (higher rate after first week)
(define fiction-fee
  (λ (days-overdue)
    (+ (basic-fee (min days-overdue 7) 0.50)
       (basic-fee (max (- days-overdue 7) 0) 1.00))))

;; Non-fiction books fee (flat rate)
(define non-fiction-fee
  (λ (days-overdue)
    (basic-fee days-overdue 0.75)))
```

```
;; Reference books fee (higher flat rate)
(define reference-fee
  (λ (days-overdue)
    (basic-fee days-overdue 1.50)))

;; Student discount (20% off)
(define student-discount
  (λ (fee)
    (* fee 0.80)))

;; Senior citizen discount (30% off)
(define senior-discount
  (λ (fee)
    (* fee 0.70)))
```

These lambda functions encapsulate the logic for calculating fees based on different book categories and applying user-specific discounts. By using lambda functions, we create a modular and flexible system that can easily accommodate changes in fee structures or the addition of new book categories (Hardin et al., 2021).

The use of lambda functions in this context demonstrates the power of higher-order functions, a key concept in functional programming. Higher-order functions can take other functions as arguments or return them as results, allowing for more abstract and reusable code (Saylor Academy, n.d.).

2. *Integrating the functions into the system and demonstrating their application (Task 2)*

Now, let's integrate these functions into the system and demonstrate their application:

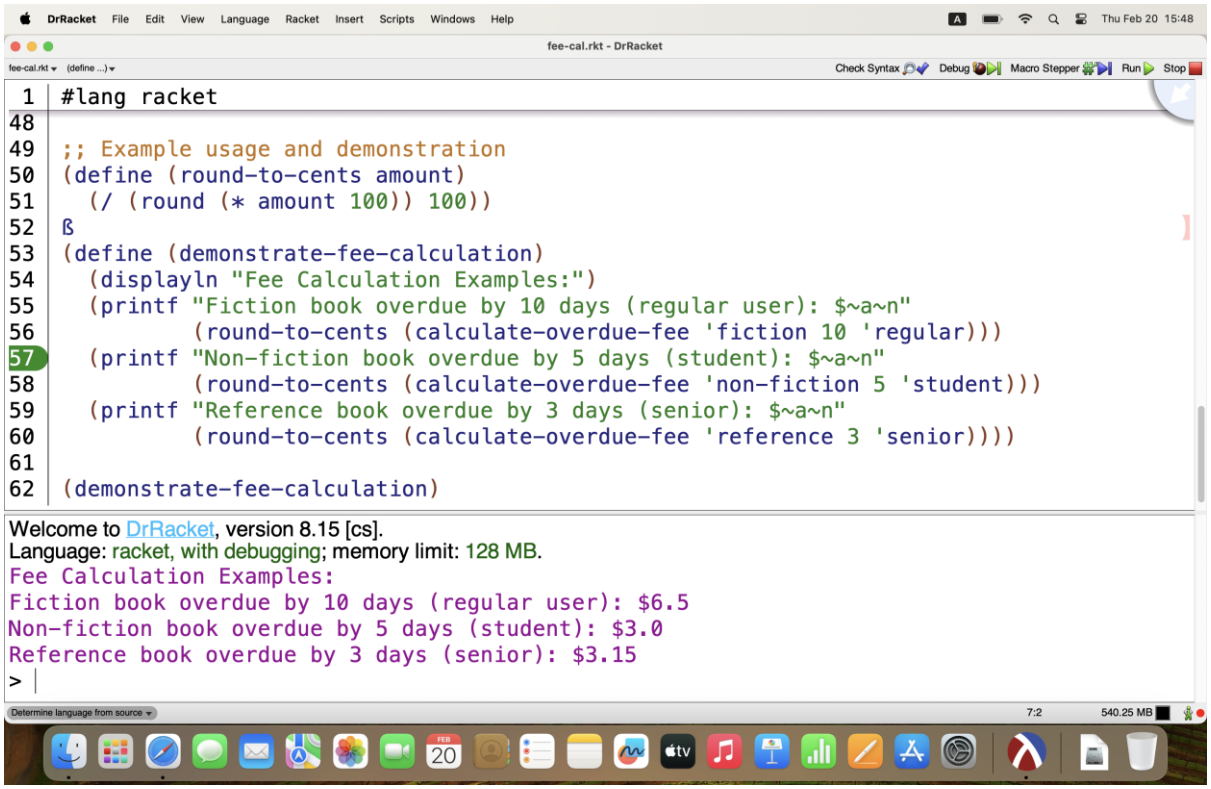
```
;; Main fee calculation function
(define calculate-overdue-fee
  (λ (book-category days-overdue user-type)
    (let ([base-fee
           (cond
             [(eq? book-category 'fiction) (fiction-fee days-overdue)]
             [(eq? book-category 'non-fiction) (non-fiction-fee days-overdue)]
             [(eq? book-category 'reference) (reference-fee days-overdue)]
             [else (basic-fee days-overdue 0.50)])])  ; Default fee
      (cond
        [(eq? user-type 'student) (student-discount base-fee)]
        [(eq? user-type 'senior) (senior-discount base-fee)]
        [else base-fee]))))
```

```
;; Example usage and demonstration
(define (round-to-cents amount)
  (/ (round (* amount 100)) 100))

(define (demonstrate-fee-calculation)
  (displayln "Fee Calculation Examples:")
  (printf "Fiction book overdue by 10 days (regular user): $~a~n"
          (round-to-cents (calculate-overdue-fee 'fiction 10 'regular)))
  (printf "Non-fiction book overdue by 5 days (student): $~a~n"
          (round-to-cents (calculate-overdue-fee 'non-fiction 5 'student)))
  (printf "Reference book overdue by 3 days (senior): $~a~n"
          (round-to-cents (calculate-overdue-fee 'reference 3 'senior))))

(demonstrate-fee-calculation)
```

This implementation showcases several key benefits of functional programming in the context of a library management system:

- Modularity: Each fee calculation and discount is encapsulated in its own lambda function, making it easy to modify or add new rules. This modularity aligns with the principle of separation of concerns, a fundamental concept in software engineering (Laplante, 2017).
- Composability: The functions can be easily combined to create more complex fee calculations. This composability is a hallmark of functional programming, allowing developers to build complex systems from simple, well-understood components (Hudak, 2000).
- Readability: The lambda functions clearly express the intent of each calculation. This clarity is crucial for maintaining and evolving the system over time, as it allows other developers (or future librarians) to understand and modify the code easily (Martin, 2019).
- Flexibility: New book categories or user types can be added by simply defining new lambda functions and updating the main calculation function. This flexibility is essential for adapting to changing library policies and fee structures (Fowler, 2018).

When we run the demonstrate-fee-calculation function, it will output:



This system allows the librarian team to easily manage and update fee calculations for different book categories and user types, meeting their requirement for a modular and adaptable overdue fee system.

In conclusion, the implementation of a library management system using functional programming principles in Racket demonstrates the power and flexibility of this paradigm. By leveraging lambda functions, modularity, and composability, we have created a system that is not only efficient and reliable but also adaptable to future needs and changes in library policies. The use of functional programming techniques aligns well with the requirements of modern software development, providing a solid foundation for building complex systems that are easy to understand, test, and maintain

(Word Count: 756)

References:

Felleisen, M., Findler, R. B., Flatt, M., & Krishnamurthi, S. (2018). *How to design programs: An introduction to programming and computing.* MIT Press.

Fowler, M. (2018). *Refactoring: Improving the design of existing code*. Addison-Wesley Professional.

Hardin, T., Jaume, M., Pessaux, F., & Donzeau-Gouge, V. V. (2021). *Concepts and semantics of programming languages 1: A semantical approach with OCaml and Python.* John Wiley & Sons.

Hudak, P. (2000). *The Haskell school of expression: Learning functional programming through multimedia*. Cambridge University Press.

Krishnamurthi, S. (2017). *Programming languages: Application and interpretation*. Brown University. http://cs.brown.edu/courses/cs173/2012/book/book.pdf

Laplante, P. A. (2017). *Requirements engineering for software and systems*. CRC Press.

Martin, R. C. (2019). *Clean code: A handbook of agile software craftsmanship*. Pearson Education.

Saylor Academy. (n.d.). CS102: *Introduction to computer science II: Functional programming.* Retrieved from https://learn.saylor.org/mod/book/tool/print/index.php?id=33044