Using Claude 3.7 Sonnet:

Create the basic structure for a text adventure game in Racket using a purely functional programming style. Include data structures for game state, rooms, and exits between rooms.

Implement an inventory system for a text adventure game that allows players to pick up, drop, and check their inventory. Show how these actions transform the game state.

```
;; ---- GAME STATE STRUCTURE ----
;; The game state is represented as a structure with the following fields:
;; - current-room: The ID of the room the player is currently in
;; - inventory: A list of items the player is carrying
;; - rooms: An immutable hash table mapping room IDs to room structures
;; - unlocked-exits: A list of unlocked exit paths (represented as [from-room to-room])
;; - game-over: A boolean indicating whether the game is over (success or failure)
;; - message: A message to display to the player

(struct game-state (current-room inventory rooms unlocked-exits game-over message)
#:transparent)

;; Room structure
(struct room (name description items exits) #:transparent)

;; Exit structure (represents a connection between rooms)
(struct exit (direction target key-item description locked-msg) #:transparent)
;; ---- GAME LOOP AND COMMAND PARSING ----

;; Parse a command string and perform the corresponding action
(define (parse-command game command-str)
  (let* ([cmd-parts (string-split (string-downcase command-str))]
         [process-item-name (lambda (parts) (string-join (cdr parts) " "))])
    (if (empty? cmd-parts)
        (struct-copy game-state game [message "Please enter a command."])
        (let ([cmd (first cmd-parts)])
          (cond
            [(or (string=? cmd "quit") (string=? cmd "exit"))
             (struct-copy game-state game [game-over #t] [message "Thanks for playing!"])]

            [(string=? cmd "look")
             (look game)]

            [(string=? cmd "inventory")
             (inventory game)]

            [(string=? cmd "take")
             (if (>= (length cmd-parts) 2)
               (take game (process-item-name cmd-parts))
               (struct-copy game-state game [message "Take what?"]))]

            [(string=? cmd "drop")
             (if (>= (length cmd-parts) 2)
               (drop game (process-item-name cmd-parts))
               (struct-copy game-state game [message "Drop what?"]))]

            [(or (string=? cmd "go") (string=? cmd "move"))
             (if (>= (length cmd-parts) 2)
               (move game (string->symbol (second cmd-parts)))
               (struct-copy game-state game [message "Go where?"]))]

            [(member cmd '("north" "south" "east" "west" "up" "down"))
             (move game (string->symbol cmd))]

            [(string=? cmd "help")
             (struct-copy game-state game
                       [message "Available commands:\n- look: Look around\n- inventory: Check your
inventory\n- take [item]: Pick up an item\n- drop [item]: Drop an item\n- go [direction]: Move in a
direction\n- north, south, east, west: Move in that direction\n- quit: Exit the game"])]

            [else (struct-copy game-state game [message "I don't understand that command."])])))))
```

Create a command parser for a text adventure game that handles commands like "look", "take item", "drop item", "go north", etc. Use pattern matching or conditional logic in a functional style.

```
;; ---- GAME ACTIONS ----

;; Look around the current room
(define (look game)
  (let* ([current-room-id (game-state-current-room game)]
         [rooms (game-state-rooms game)]
         [current-room (hash-ref rooms current-room-id)]
         [items-str (if (empty? (room-items current-room))
                      "There are no items here."
                      (string-append "You see: " (string-join (room-items current-room) ", ")))]
         [exits-str (string-join
                     (map (lambda (exit)
                            (exit-description exit))
                          (room-exits current-room))
                     "\n")]
         [message (string-append
                   "--- " (room-name current-room) " ---\n\n"
```

```
                  (room-description current-room) "\n\n"
                  items-str "\n\n"
                  "Available exits:\n" exits-str)])
    (struct-copy game-state game [message message])))

;; Take an item from the current room
(define (take game item-name)
  (let* ([current-room-id (game-state-current-room game)]
         [rooms (game-state-rooms game)]
         [current-room (hash-ref rooms current-room-id)])
    (if (member item-name (room-items current-room))
        ;; Item exists in room, take it
        (let* ([new-room-items (remove item-name (room-items current-room))]
               [new-room (struct-copy room current-room [items new-room-items])]
               [new-rooms (hash-set rooms current-room-id new-room)]
               [new-inventory (cons item-name (game-state-inventory game))])
          (struct-copy game-state game
                       [inventory new-inventory]
                       [rooms new-rooms]
                       [message (format "You picked up the ~a." item-name)]))
        ;; Item doesn't exist in room
        (struct-copy game-state game
                     [message (format "There is no ~a here." item-name)]))))

;; Drop an item from inventory into the current room
(define (drop game item-name)
  (let* ([current-room-id (game-state-current-room game)]
         [rooms (game-state-rooms game)]
         [current-room (hash-ref rooms current-room-id)])
    (if (member item-name (game-state-inventory game))
        ;; Item exists in inventory, drop it
        (let* ([new-inventory (remove item-name (game-state-inventory game))]
               [new-room-items (cons item-name (room-items current-room))]
               [new-room (struct-copy room current-room [items new-room-items])]
               [new-rooms (hash-set rooms current-room-id new-room)])
          (struct-copy game-state game
                       [inventory new-inventory]
                       [rooms new-rooms]
                       [message (format "You dropped the ~a." item-name)]))
        ;; Item doesn't exist in inventory
        (struct-copy game-state game
                     [message (format "You don't have a ~a." item-name)]))))

;; Check inventory
```

```
(define (inventory game)
  (let ([inv (game-state-inventory game)])
    (if (empty? inv)
        (struct-copy game-state game [message "Your inventory is empty."])
        (struct-copy game-state game
                    [message (string-append "Inventory: " (string-join inv ", "))]))))

;; Helper function to check if an exit is unlocked
(define (exit-unlocked? game from-room to-room)
  (member (list from-room to-room) (game-state-unlocked-exits game)))

;; Try to move in a direction
(define (move game direction)
  (let* ([current-room-id (game-state-current-room game)]
         [rooms (game-state-rooms game)]
         [current-room (hash-ref rooms current-room-id)]
         [possible-exit (findf (lambda (exit) (eq? (exit-direction exit) direction))
                               (room-exits current-room))])
    (cond
      [(not possible-exit)
       (struct-copy game-state game
                    [message (format "You can't go ~a from here." direction)])]

      ;; Check if the exit is already unlocked
      [(exit-unlocked? game current-room-id (exit-target possible-exit))
       (let ([new-room (exit-target possible-exit)])
         (if (eq? new-room 'vault)
             ;; Winning condition - reached the vault
             (struct-copy game-state game
                          [current-room new-room]
                          [message (string-append
                                    (room-description (hash-ref rooms new-room))
                                    "\n\nCONGRATULATIONS! You've successfully broken into the vault!")]
                          [game-over #t])
             ;; Normal room transition
             (struct-copy game-state game
                          [current-room new-room]
                          [message (string-append
                                    "You move " (symbol->string direction) ".\n\n"
                                    "--- " (room-name (hash-ref rooms new-room)) " ---\n\n"
                                    (room-description (hash-ref rooms new-room)))])))]

      ;; Check if the exit requires a key and player doesn't have it
      [(and (exit-key-item possible-exit)
        (not (has-required-items? game (exit-key-item possible-exit))))
     (struct-copy game-state game
             [message (exit-locked-msg possible-exit)])]

    ;; Player has the key, unlock the door and proceed
    [else
     (let* ([new-room (exit-target possible-exit)]
            ;; Add this exit to unlocked-exits list if it has a key requirement
            [new-unlocked-exits (if (exit-key-item possible-exit)
                                    (cons (list current-room-id new-room)
                                          (game-state-unlocked-exits game))
                                    (game-state-unlocked-exits game))]
            [unlock-msg (if (exit-key-item possible-exit)
                            (string-append
                             "You unlock the door with "
                             (if (string? (exit-key-item possible-exit))
                                 (string-append "the " (exit-key-item possible-exit))
                                 (string-append "your items"))
                             ". ")
                            "")])
       (if (eq? new-room 'vault)
           ;; Winning condition - reached the vault
           (struct-copy game-state game
                    [current-room new-room]
                    [unlocked-exits new-unlocked-exits]
                    [message (string-append
                              unlock-msg
                              (room-description (hash-ref rooms new-room))
                              "\n\nCONGRATULATIONS! You've successfully broken into the vault!")]
                    [game-over #t])
           ;; Normal room transition
           (struct-copy game-state game
                    [current-room new-room]
                    [unlocked-exits new-unlocked-exits]
                    [message (string-append
                              unlock-msg
                              "You move " (symbol->string direction) ".\n\n"
                              "--- " (room-name (hash-ref rooms new-room)) " ---\n\n"
                              (room-description (hash-ref rooms new-room)))])))])))
```

Implement a system for providing hints when the player appears stuck, such as if they've been in the same room for many turns.

```
 (vault-door . ,(room "Vault Entrance"
            "You're standing in front of the massive bank vault door.\nIt has both a keypad for a
code and requires a special key."
            (list "vault key")
            (list (exit 'north 'lobby #f "The lobby is to the north." #f)
                 (exit 'south 'vault (list "vault code" "vault key") "The vault is beyond this door."
                      "The vault door is locked. You need both the vault code and the vault key to
open it."))))
```

Design a win/lose condition system for a text adventure. In this bank heist game, define what constitutes winning (reaching the vault) and possible losing scenarios.

```
;; Check if the player has all required items
(define (has-required-items? game required-items)
  (let ([inventory (game-state-inventory game)])
    (cond
      [(string? required-items) (member required-items inventory)]
      [(list? required-items) (andmap (lambda (item) (member item inventory)) required-items)]
      [else #t])))
```

Finally, design the game loop using all the functions you made.

```
;; Main game loop
(define (game-loop)
  (let loop ([game (new-game)])
    (displayln (game-state-message game))
    (newline)

    (when (not (game-state-game-over game))
      (display "> ")
      (flush-output)
      (let* ([input (read-line)]
             [new-game (parse-command game input)])
        (newline)
        (loop new-game)))))

;; Start the game
(game-loop)
```

I need so when a door is unlocked it stays unlocked

```
;; Helper function to check if an exit is unlocked
(define (exit-unlocked? game from-room to-room)
  (member (list from-room to-room) (game-state-unlocked-exits game)))
```

Reflection:

The only problems I had with this were using ChatGPT and DeepSeek, since they would give me such complicated code — and often it wouldn't even work. They didn't understand the specifics or had their own ideas of how to use structs, plus there's the memory limit of ChatGPT free. When I asked Claude the same prompts I gave GPT, it gave a really good response and sometimes even coded the whole thing, but not always with the specifics I wanted.

What I ended up doing was making the rooms myself, showing them to Claude, and then asking how to make the functions for things like take, look, move, drop, etc. These LLM tools definitely help us tackle bigger projects, but I still did extensive testing to make sure there were no errors. We just have to make sure the code actually works, because what I've learned is that LLMs don't always give the best code.

My experience with DeepSeek was that it often left a bunch of parentheses in the wrong places. Over the whole semester, I think using LLMs has honestly helped me learn more advanced concepts and also helped me get through my class projects a lot faster, since I only had to check over the code instead of writing everything from scratch.

The process basically goes like this: figure out a prompt, choose the model you want to use, read over the code, implement it, and make sure it follows the rubric. Overall, using LLMs has helped me understand what coding is going to look like in the future and how to adapt my knowledge to work with these tools.