***Solved 2023 First Exam Past Paper***

**Q1:**

Answer the following questions in 1-2 short sentences!

1. **What does the term shadowing mean?**

**Answer**: Shadowing happens when a variable declared in a local scope (like a function or block) has the same name as a variable in a higher scope, making the local variable take precedence over the outer one within that scope.

**Detailed Answer:**

#lang racket

(define x 10) ; Outer variable

(define (foo)

(define x 20) ; Inner variable shadows the outer variable

x) ; Returns 20

(displayln (foo)) ; Prints 20

(displayln x) ; Prints 10

In this code:

* The outer variable x is set to 10.
* Inside the function foo, a new variable x is defined and set to 20, shadowing the outer x.
* Calling (foo) returns 20, the value of the inner x.
* Outside the function, x retains its value of 10, so (displayln x) prints 10.

1. **What must be taken into account when evaluating the order of cond expressions?**

**Answer:** When evaluating cond expressions in Racket, predicates are checked sequentially from top to bottom, stopping at the first true predicate. Ensure specific conditions come before general ones and consider performance and side effects.

**Detailed Answer:**

When evaluating the order of cond expressions in a language like Racket (or Scheme), the following considerations must be taken into account:

1. **Sequential Evaluation**: cond expressions are evaluated sequentially from top to bottom. This means the first condition that evaluates to true will trigger its corresponding expression to be executed, and no subsequent conditions will be checked.
2. **Order Matters**: The order of conditions is crucial because the first true condition encountered will stop further evaluation. If there are overlapping or related conditions, their order can change the outcome of the program.
3. **Specificity First**: Place more specific conditions before more general ones. If a more general condition comes first, it might catch cases that should be handled by a more specific condition.
4. **Efficiency**: Place the most likely true conditions earlier to minimize unnecessary checks. This can improve the efficiency of your program, especially if evaluating conditions is costly.
5. **Default Case**: It is common practice to have a default case (often written as [else expr]) at the end of the cond expression to handle any situations not covered by the previous conditions. This ensures that the cond expression always produces a result.

#lang racket

(define (classify-number n)

(cond

[(< n 0) 'negative] ; Specific case for negative numbers

[(= n 0) 'zero] ; Specific case for zero

[(> n 0) 'positive] ; Specific case for positive numbers

[else 'unknown])) ; Default case

(displayln (classify-number -5)) ; Prints 'negative

(displayln (classify-number 0)) ; Prints 'zero

(displayln (classify-number 10)) ; Prints 'positive

In this example:

* The conditions are checked sequentially.
* The specific cases for negative numbers and zero are placed before the positive numbers.
* The else clause acts as a default case to ensure a result is always returned, though in this specific example it is not necessary because all possible cases are covered.

By considering these factors, you ensure that your cond expressions are evaluated correctly and efficiently.

1. **What does structural recursion mean?**

**Answer:** Structural recursion means solving a problem by breaking it down into smaller parts of the same type, using a function that calls itself on these smaller parts until a base case is reached. This approach follows the natural structure of the data being processed.

**Detailed Answer:**

Structural recursion is a form of recursion where a function solves a problem by recursively calling itself on smaller or simpler instances of the same data structure. The structure of the recursion closely follows the structure of the data being processed, typically involving base cases that handle the simplest forms of the data and recursive cases that break down complex data into simpler parts.

Here is a short example in Racket:

**#lang racket**

**(define (sum-list lst)**

**(cond**

**[(empty? lst) 0] ; Base case: empty list**

**[else (+ (first lst) (sum-list (rest lst)))])) ; Recursive case: sum the first element and the sum of the rest**

**(displayln (sum-list '(1 2 3 4 5))) ; Prints 15**

In this example:

* The sum-list function uses structural recursion to sum the elements of a list.
* The base case handles an empty list, returning 0.
* The recursive case breaks the list into its first element and the rest, summing the first element with the result of recursively calling sum-list on the rest of the list.

1. **What does the term atom mean in Prolog? Give an example!**

**Answer:** In Prolog, an **atom** refers to a basic data element representing a constant symbol or name. It is typically used to denote predicates, constants, or variables starting with a lowercase letter or enclosed in single quotes.

**Example**:

**likes(john, pizza).**

In this example, john and pizza are atoms representing names or constants within Prolog predicates.

1. **What is a scope in programming?**

**Answer:** In declarative programming, **scope** determines where variables, functions, and other elements can be accessed and used within the program. It defines the boundaries within which identifiers are valid and visible.

1. **Briefly explain the term "accumulator invariant".**

**Answer:** In Racket, an **accumulator invariant** in recursive functions typically involves using an accumulator parameter to maintain and update a cumulative result or state throughout recursive calls. This approach is fundamental for iterative processes in functional programming without resorting to mutable state.

**Detailed Answer:**

Here's an example illustrating an accumulator invariant in a recursive function that calculates the sum of a list of numbers:

#lang racket

(define (sum-list lst)

(sum-list-helper lst 0))

(define (sum-list-helper lst acc)

(if (null? lst)

acc ; Base case: return the accumulator when the list is empty

(sum-list-helper (cdr lst) (+ acc (car lst))))) ; Recursive case: accumulate the sum of list elements

; Example usage

(displayln (sum-list '(1 2 3 4 5))) ; Output: 15

Explanation:

* **sum-list** function computes the sum of a list lst.
* **sum-list-helper** is a helper function with two arguments: lst (current list being processed) and acc (accumulator).
* When lst is empty ((null? lst)), the base case is triggered, and it returns the accumulator acc, which holds the sum of list elements.
* Otherwise ((sum-list-helper (cdr lst) (+ acc (car lst)))), it recursively calls itself with the rest of the list (cdr lst) and updates the accumulator acc by adding the current element of the list (car lst).
* This ensures that acc accumulates the sum of list elements as the function recursively processes each element.

In this example:

* The accumulator acc maintains the invariant that it accumulates the sum of list elements processed so far.
* This demonstrates how to use an accumulator invariant in a simple recursive function in Racket to compute a result iteratively while adhering to functional programming principles.

1. **What is syntactic sugar?**

**Answer**: In declarative programming, **syntactic sugar** refers to language features that provide more convenient or readable ways to express code without changing its underlying functionality. It enhances clarity and reduces verbosity, making code easier to write and understand.

**Detailed Answer:**

**#lang racket**

**; Without syntactic sugar**

**(define (add1 x)**

**(+ x 1))**

**; With syntactic sugar (using let)**

**(define (add1-sugar x)**

**(let ((result (+ x 1)))**

**result))**

**; Example usage**

**(displayln (add1 5)) ; Output: 6**

**(displayln (add1-sugar 5)) ; Output: 6**

Explanation:

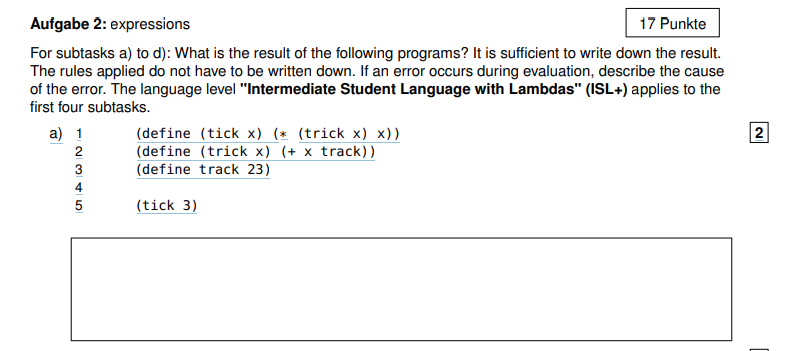
* The function add1 adds 1 to its argument x using the traditional define syntax.
* The function add1-sugar achieves the same result using syntactic sugar with a let binding, where result is bound to the value of (+ x 1).

In this example:

* Both add1 and add1-sugar produce the same output (6 for input 5).
* Syntactic sugar with let simplifies the process of binding variables (result in this case) and makes the code more readable by reducing the need for explicit lambda expressions or additional function definitions.

This demonstrates how syntactic sugar in Racket can improve code clarity and conciseness by providing convenient shortcuts for common programming patterns.

**Q2:**

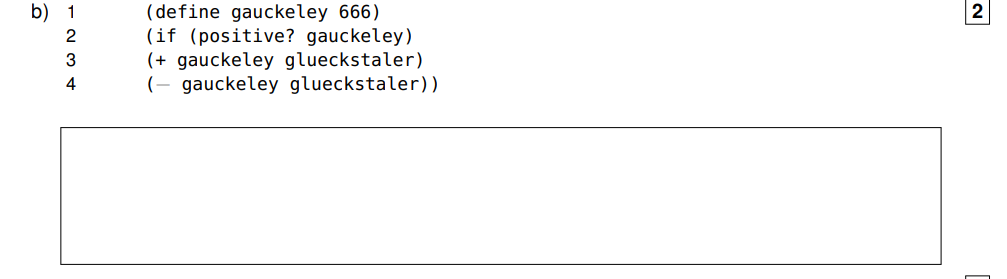
****

**Answer: 78**

**Detailed Explanation:**

1. **Defining tick**: The function tick is defined as (\* (trick x) x). This means it calls trick with argument x and multiplies the result by x.
2. **Defining trick**: The function trick is defined as (+ x track), where track is a variable.
3. **Defining track**: track is set to 23.
4. **Evaluating (tick 3)**:
   * tick is called with x = 3.
   * Inside tick, trick is called with x = 3.
5. **Evaluating (trick 3)**:
   * trick is defined as (+ x track).
   * Substituting x = 3 and track = 23, we get (+ 3 23).
   * This simplifies to 26.
6. **Completing the evaluation of (tick 3)**:
   * Now tick evaluates to (\* (trick 3) 3), which is (\* 26 3).
   * Calculating this product gives 78.

Therefore, the result of (tick 3) is 78.



**Answer:** glueckstaler: this variable is not defined in: glueckstaler

**Detailed Explanation:**

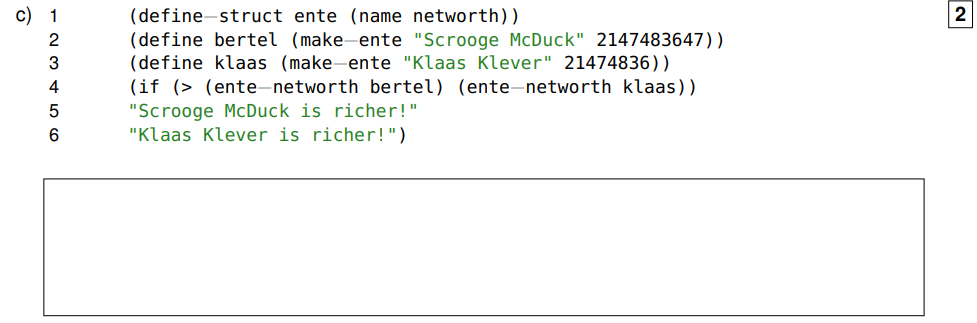
Result: The result depends on the value of glueckstaler, which is not provided in the given code snippet. The expression will evaluate to either 666 + glueckstaler or 666 - glueckstaler, depending on whether gauckeley is positive or not.

If glueckstaler were, for example, 10, the result would be 676 (assuming gauckeley is positive).

If glueckstaler were 20, the result would be 646 (assuming gauckeley is positive).

The exact numerical result cannot be determined without knowing the value of glueckstaler.

If gauckeley were not positive, the result would be -glueckstaler, which similarly depends on the value of glueckstaler.



**Answer:** "Scrooge McDuck is richer!"

**Detailed Explanation:**

 ente is a defined structure with fields name and networth.

 bertel and klaas are instances of ente, initialized with specific values for name and networth.

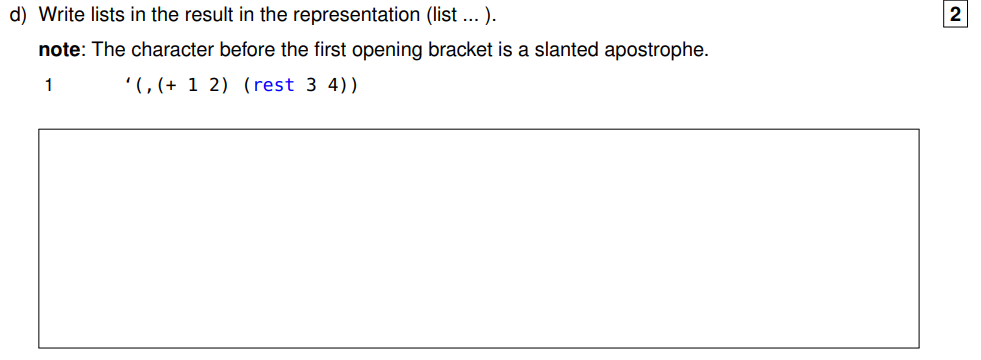
 The if statement compares the networth of bertel and klaas.

 bertel has a networth of 2147483647 (assuming it's within the range of Scheme's integers).

 klaas has a networth of 21474836.

 The condition (> (ente-networth bertel) (ente-networth klaas)) evaluates to true because 2147483647 is greater than 21474836.

 Therefore, the result of the if expression is "Scrooge McDuck is richer!".

****

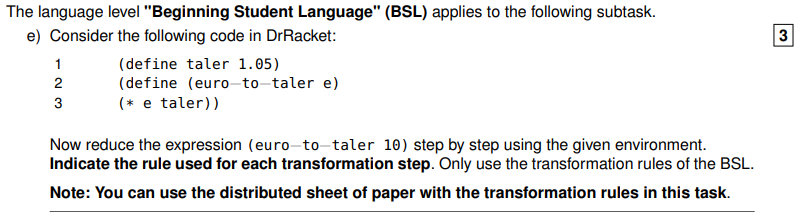
**Answer:** (list 3 (list 'rest 3 4))

(list (list 'string-append "Drake" 'Mallard) (list 'Darkwing))

**Detailed Explanation:**

* (+ 1 2) evaluates to 3.
* (rest 3 4) is represented as (list 'rest 3 4), where 'rest is quoted to ensure it is treated as a symbol.
* Therefore, '(+ 1 2) is replaced by 3 and 'rest 3 4 remains as (list 'rest 3 4).

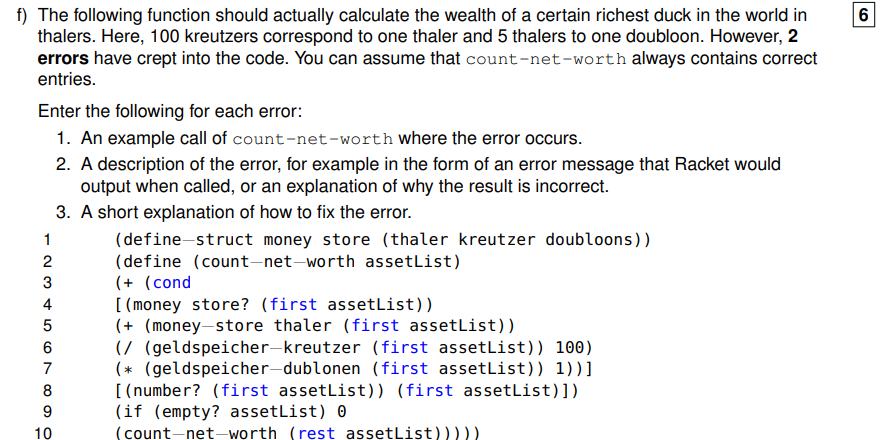
So, the result in the representation (list ...) is (list 3 (list 'rest 3 4)).

****

**Answer:**

So, the step-by-step reduction is as follows:

1. (euro−to−taler 10)
2. (\* 10 taler) (by function application rule)
3. (\* 10 1.05) (by variable substitution rule)
4. 10.5 (by arithmetic operation rule)



**Error 1: Incorrect Struct Field Accessor Names**

1. **Example Call:** (count−net−worth (list (money store 10 500 2)))
2. **Description:** The error occurs because the accessor names are incorrect. Racket will raise an error like: unbound identifier in module in: money-store-thaler.
3. **Fix:** Correct the accessor names to match the struct definition. Use money-store-thaler, money-store-kreutzer, and money-store-doubloons instead of the incorrect names.



(define (count−net−worth assetList)

(+ (cond

[(money store? (first assetList))

(+ (money-store-thaler (first assetList))

(/ (money-store-kreutzer (first assetList)) 100)

(\* (money-store-doubloons (first assetList)) 5))]

[(number? (first assetList)) (first assetList)])

(if (empty? assetList) 0

(count−net−worth (rest assetList)))))

**Error 2: Incorrect Base Case Position**

1. **Example Call:** (count−net−worth (list))
2. **Description:** The error occurs because the base case (if (empty? assetList) 0 ...) is placed outside of the cond expression, leading to incorrect recursion.
3. **Fix:** Move the base case check outside the addition operation and place it as the first condition.



(define (count−net−worth assetList)

(if (empty? assetList)

0

(+ (cond

[(money store? (first assetList))

(+ (money-store-thaler (first assetList))

(/ (money-store-kreutzer (first assetList)) 100)

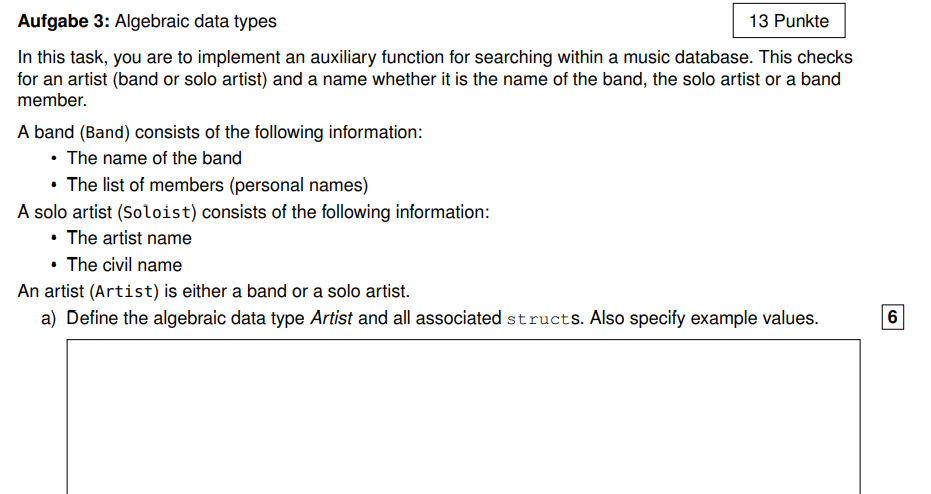
(\* (money-store-doubloons (first assetList)) 5))]

[(number? (first assetList)) (first assetList)])

(count−net−worth (rest assetList)))))

With these corrections, the function will correctly compute the net worth.

**Q3:**

****

**;; Define the struct for Band**

**(define-struct band [name members])**

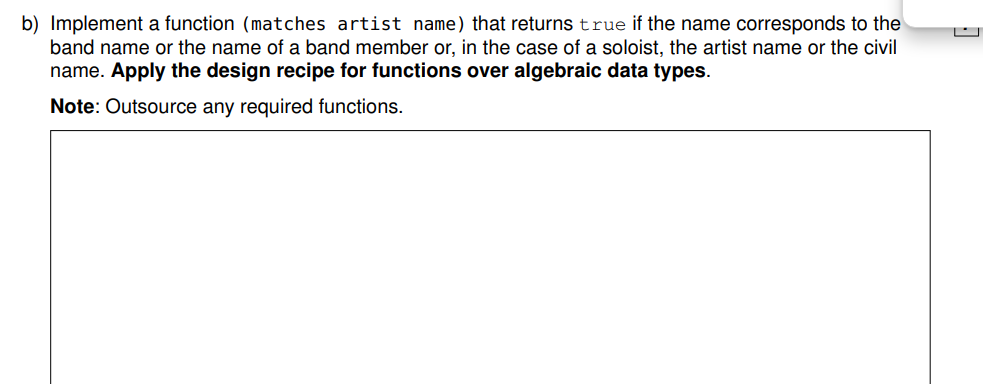
**;; Define the struct for Soloist**

**(define-struct soloist [artist-name civil-name])**

**;; Example values**

**(define band-example (make-band "The Beatles" (list "John Lennon" "Paul McCartney" "George Harrison" "Ringo Starr")))**

**(define soloist-example (make-soloist "Adele" "Adele Laurie Blue Adkins"))**

****

**;; Function to check if a name matches any element in a list of names**

**(define (name-in-list? name lst)**

**(cond**

**[(empty? lst) #f]**

**[(string=? name (first lst)) #t]**

**[else (name-in-list? name (rest lst))]))**

**;; Function to match name with artist**

**(define (matches? artist name)**

**(cond**

**[(band? artist)**

**(or (string=? name (band-name artist))**

**(name-in-list? name (band-members artist)))]**

**[(soloist? artist)**

**(or (string=? name (soloist-artist-name artist))**

**(string=? name (soloist-civil-name artist)))]))**

**;; Tests**

**(matches? band-example "The Beatles") ; should return #t**

**(matches? band-example "John Lennon") ; should return #t**

**(matches? band-example "Adele") ; should return #f**

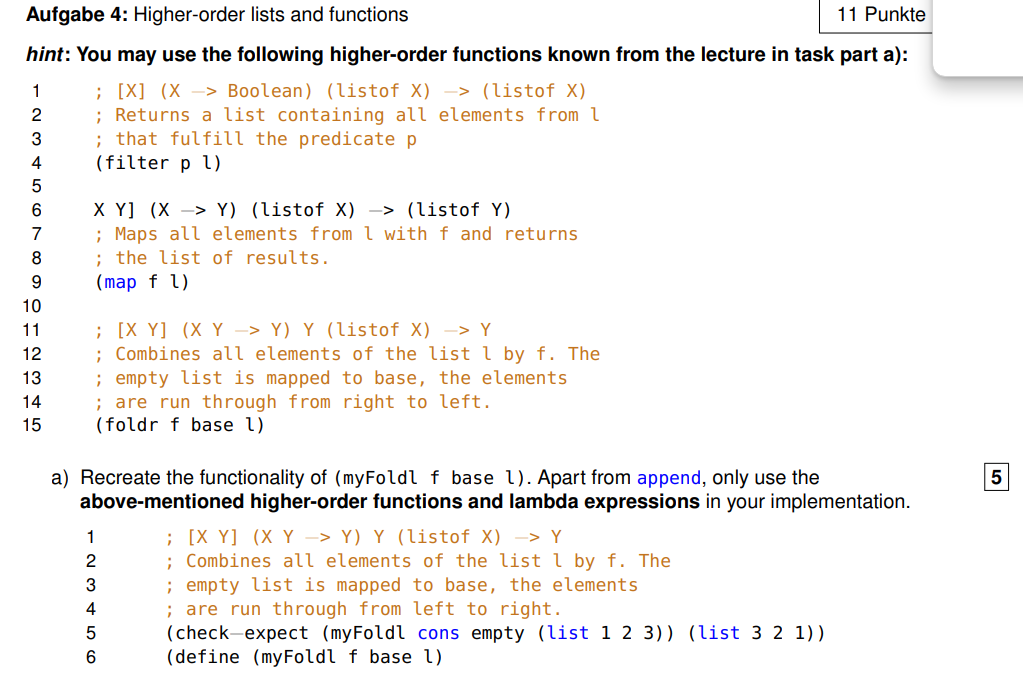
**(matches? soloist-example "Adele") ; should return #t**

**(matches? soloist-example "Adele Laurie Blue Adkins") ; should return #t**

**(matches? soloist-example "John Lennon") ; should return #f**

**Explanation:**

1. **Struct Definitions**: We define band and soloist structs with their respective fields.
2. **Example Values**: We create instances of band and soloist for testing.
3. **name-in-list? Function**: A helper function that checks if a name is in a list of names. It recursively checks each element in the list.
4. **matches? Function**: This function uses a cond expression to differentiate between band and soloist. It then checks the appropriate fields for a match using string=? and name-in-list?.
5. **Tests**: We include tests to verify our function works as expected.

****

**(define (myFoldl f base l)**

**(foldr (lambda (x acc)**

**(cons x acc)) ;; Corrected: cons x acc instead of f acc x**

**base**

**(reverse l)))**

**;; Explanation:**

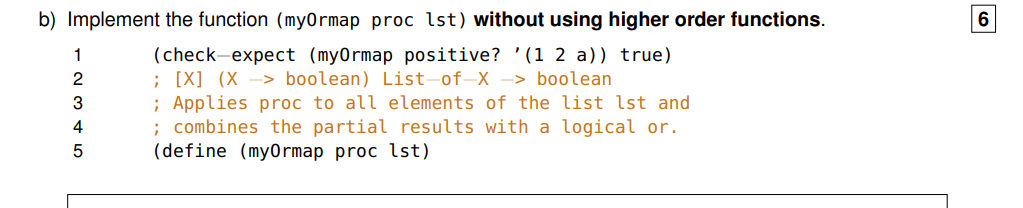
**;; - We use foldr to process the list l from right to left.**

**;; - The lambda function (lambda (x acc) (lambda (ys) (acc (f ys x)))) accumulates the result.**

**;; - base is the starting value for the accumulation.**

**;; Example usage:**

**(check-expect (myFoldl cons empty (list 1 2 3)) (list 3 2 1))**

****

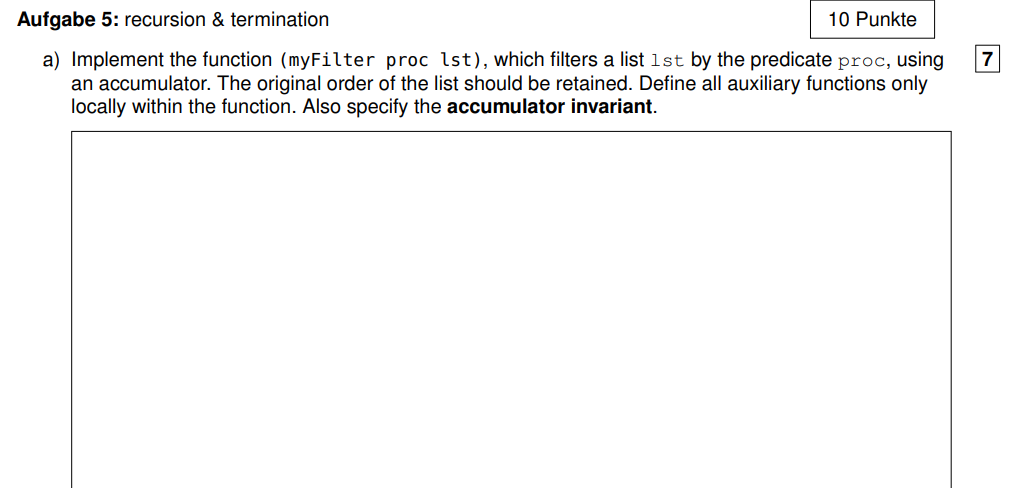
**(define (myOrmap proc lst)**

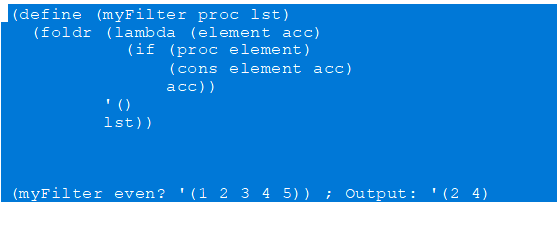
**(cond ((null? lst) #f) ; base case: empty list returns false**

**((proc (car lst)) #t) ; if proc applied to the first element is true, return true**

**(else (myOrmap proc (cdr lst))))) ; otherwise recursively check the rest of the list**

**(check-expect (myOrmap positive? '(-1 2 3)) true)**

****

****

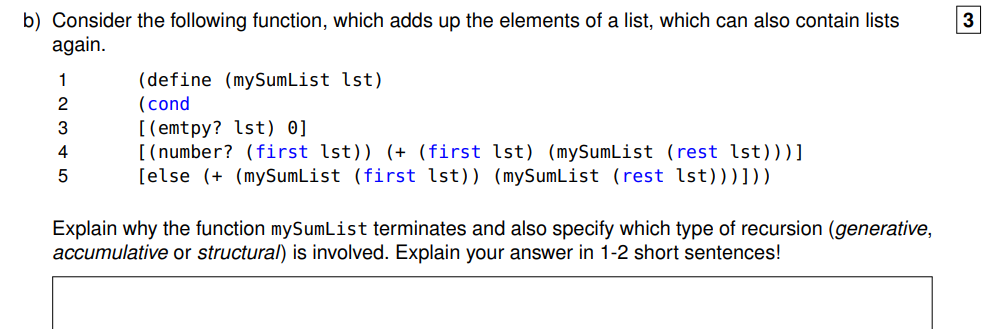
**Explanation:**

1. **Function Definition (myFilter)**:
   * myFilter is a function that takes two arguments:
     + proc: This is a predicate function that will be applied to each element of the list.
     + lst: This is the list that we want to filter based on the predicate proc.
2. **foldr Function**:
   * foldr is a higher-order function in Racket that processes lists from right to left (hence the 'r' in foldr).
   * It takes three arguments:
     + A lambda function (lambda (element acc) ...): This function takes two parameters:
       - element: The current element being processed from the list lst.
       - acc: The accumulator that collects the filtered elements.
     + An initial value for the accumulator '( ): This initializes the accumulator as an empty list.
     + The list lst: This is the list that foldr will iterate over.
3. **Lambda Function Explanation**:
   * (lambda (element acc) ...): This lambda function is applied to each element of lst by foldr.
   * Inside the lambda function:
     + (if (proc element) ...) checks if applying the predicate proc to element returns true.
     + If (proc element) is true:
       - (cons element acc): Prepends element to the accumulator acc. This means element is added to the filtered list.
     + If (proc element) is false:
       - acc: If the predicate proc returns false for element, acc remains unchanged.
4. **How foldr Works**:
   * foldr applies the lambda function (lambda (element acc) ...) to each element of lst from right to left.
   * It accumulates the results of each application of the lambda function into the accumulator, which starts as an empty list '( ).
5. **Example Usage**:
   * (myFilter even? '(1 2 3 4 5)): Here, even? is a built-in predicate function that checks if a number is even.
   * The list '(1 2 3 4 5) is passed as lst.
   * The output will be '(2 4), which contains only the even numbers from the original list, retaining their original order.

**Step-by-Step Execution:**

* **Input**: (myFilter even? '(1 2 3 4 5))
  + proc is even?, so proc element will check if each element is even.
  + lst is '(1 2 3 4 5).
* **Execution**:
  + foldr starts with the lambda function (lambda (element acc) ...).
  + For each element in '(1 2 3 4 5), it applies (if (proc element) ...):
    - For 1, (proc 1) (i.e., (even? 1)) is false, so acc remains '( ).
    - For 2, (proc 2) (i.e., (even? 2)) is true, so (cons 2 acc) adds 2 to '( ), resulting in '(2).
    - For 3, (proc 3) (i.e., (even? 3)) is false, so acc remains '(2).
    - For 4, (proc 4) (i.e., (even? 4)) is true, so (cons 4 acc) adds 4 to '(2), resulting in '(4 2).
    - For 5, (proc 5) (i.e., (even? 5)) is false, so acc remains '(4 2).
  + foldr returns '(4 2) as the final result.

Therefore, (myFilter even? '(1 2 3 4 5)) produces '(2 4), which contains only the even numbers from the original list, in their original order.

****

#### Termination:

* **Base Case**: The function has a base case [(empty? lst) 0] that returns 0 when the list is empty. This ensures that the recursion stops when there are no more elements to process.
* **Recursive Cases**:
  + When the first element is a number: [(number? (first lst)) (+ (first lst) (mySumList (rest lst)))]
    - The function processes the first element and recursively calls itself on the rest of the list (rest lst).
  + When the first element is a list: [else (+ (mySumList (first lst)) (mySumList (rest lst)))]
    - The function recursively processes both the first element (mySumList (first lst)) and the rest of the list (mySumList (rest lst)).

Each recursive call reduces the size of the input list (lst) or its sublists, eventually reaching the base case when the list is empty. This guarantees that the function will terminate.

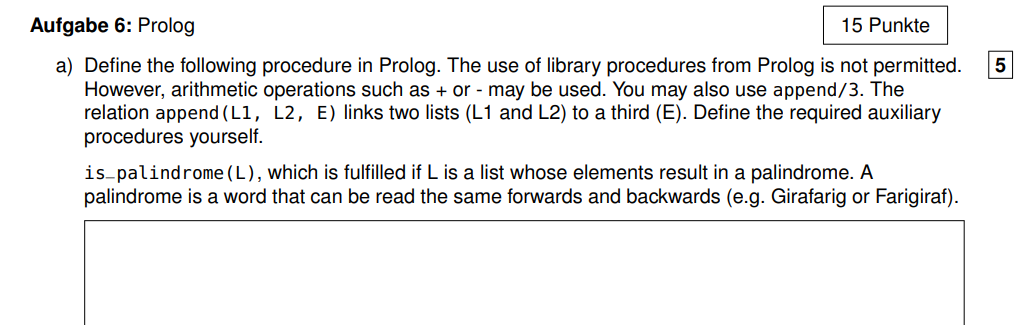
#### Type of Recursion:

* **Structural Recursion**:
  + The function mySumList is an example of structural recursion because it directly processes the structure of the input list. Each recursive call operates on smaller parts of the original list structure:
    - For a list element, it processes the element and recursively processes the rest of the list.
    - For a sublist element, it recursively processes the sublist and then the rest of the list.
* The recursion is guided by the structure of the list, breaking it down step-by-step until reaching the simplest form (an empty list).

### Summary:

* **Termination**: mySumList terminates because each recursive call works on a smaller portion of the list or sublist, ultimately reaching an empty list where the recursion stops.
* **Recursion Type**: The function uses structural recursion because it processes and recurses based on the structure of the list and its sublists.

The function mySumList terminates due to its base case for an empty list and recursive calls on smaller sublists. It uses structural recursion by processing the list's structure directly.

****

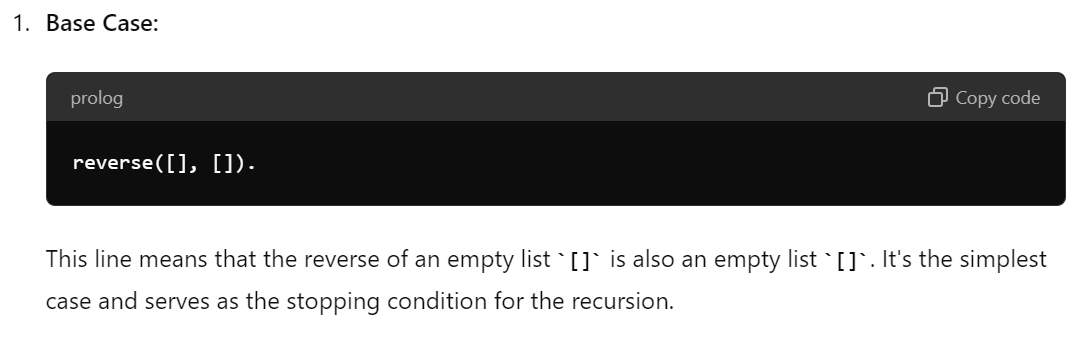
### Part (a): Defining is\_palindrome(L) in Prolog

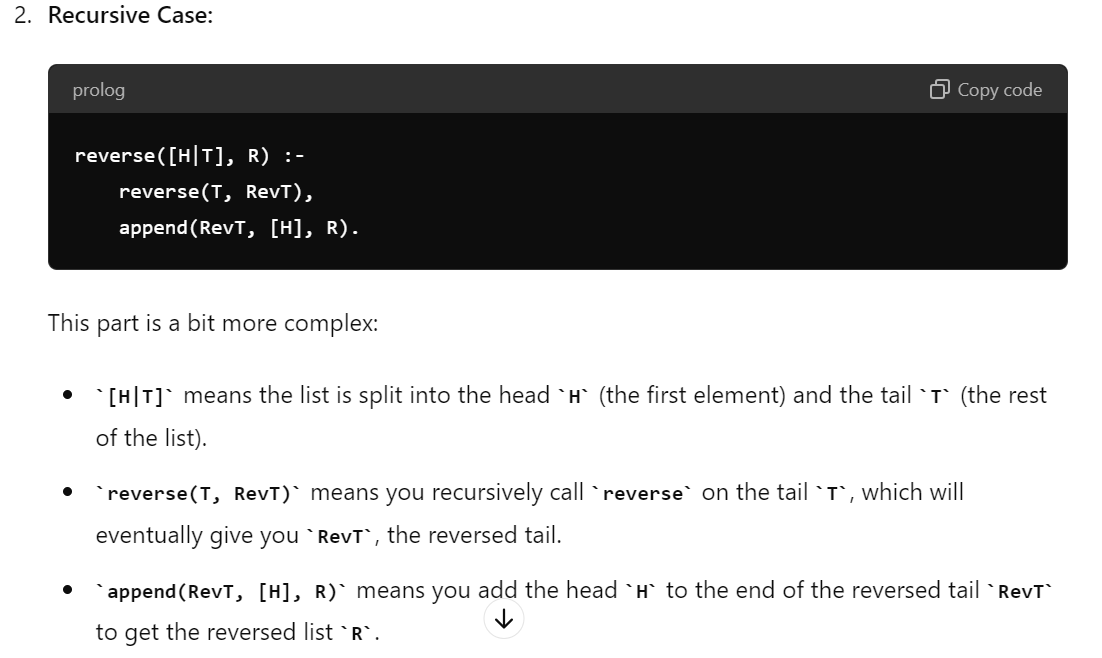
To define the is\_palindrome(L) predicate, we need to check if the list LLL is the same forwards and backwards. We can do this by comparing the list with its reverse. Here's the Prolog code:

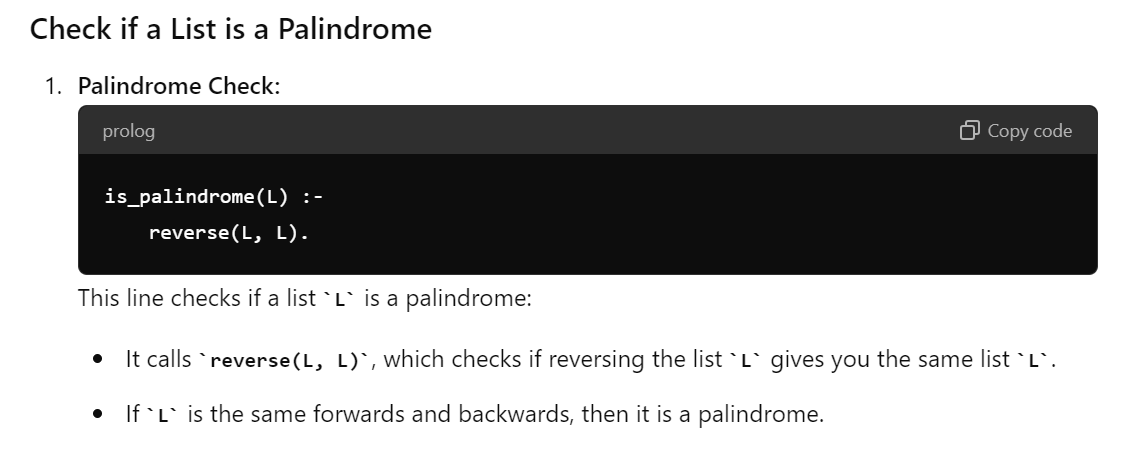
****

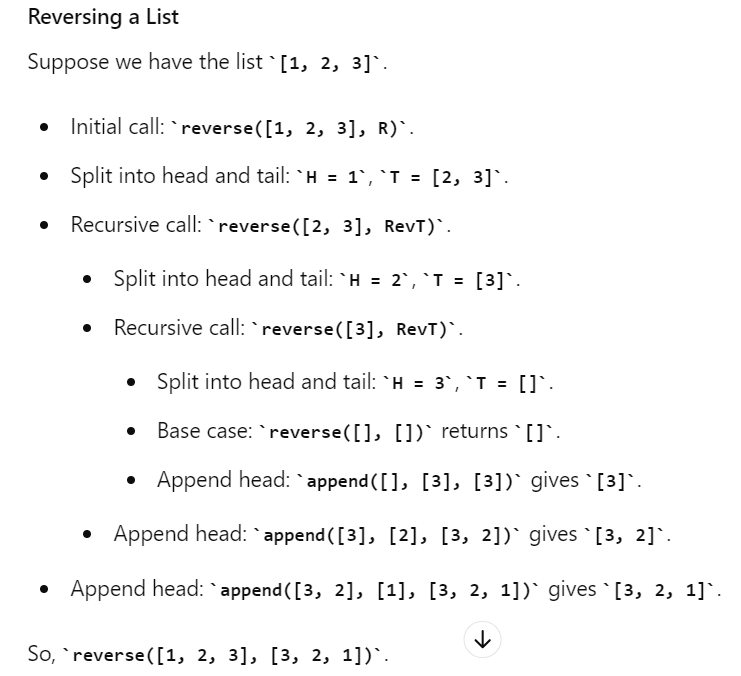
**Explanation:**

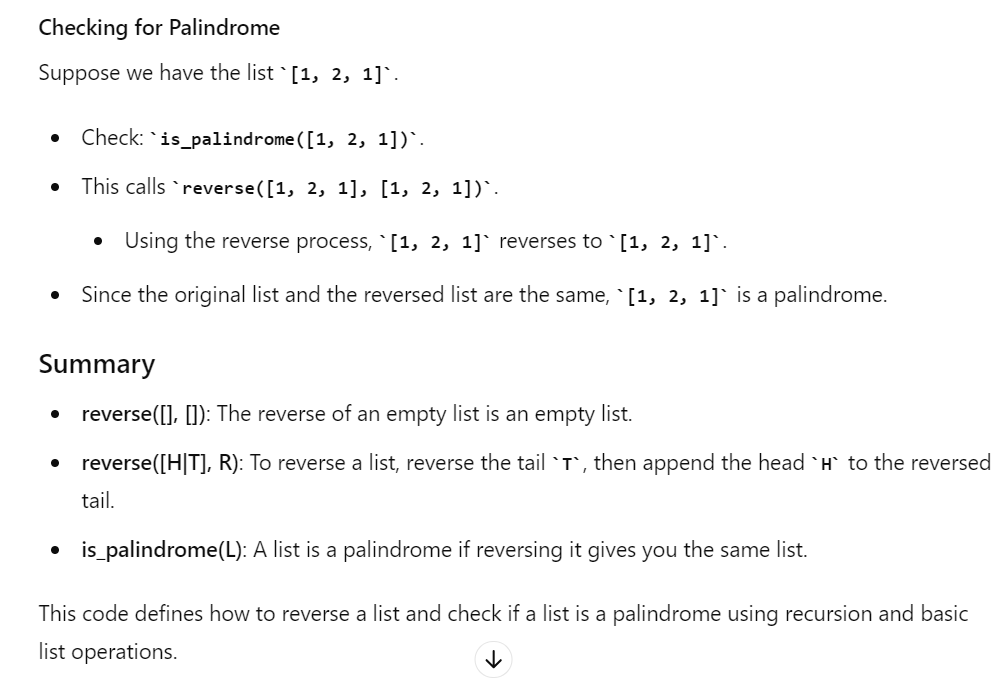
1. reverse([], []): The reverse of an empty list is an empty list.
2. reverse([H|T], R): To reverse a non-empty list, reverse the tail (T) to get RevT, then append the head (H) to the end of RevT to get the result R.
3. is\_palindrome(L): A list LLL is a palindrome if it is equal to its reverse.

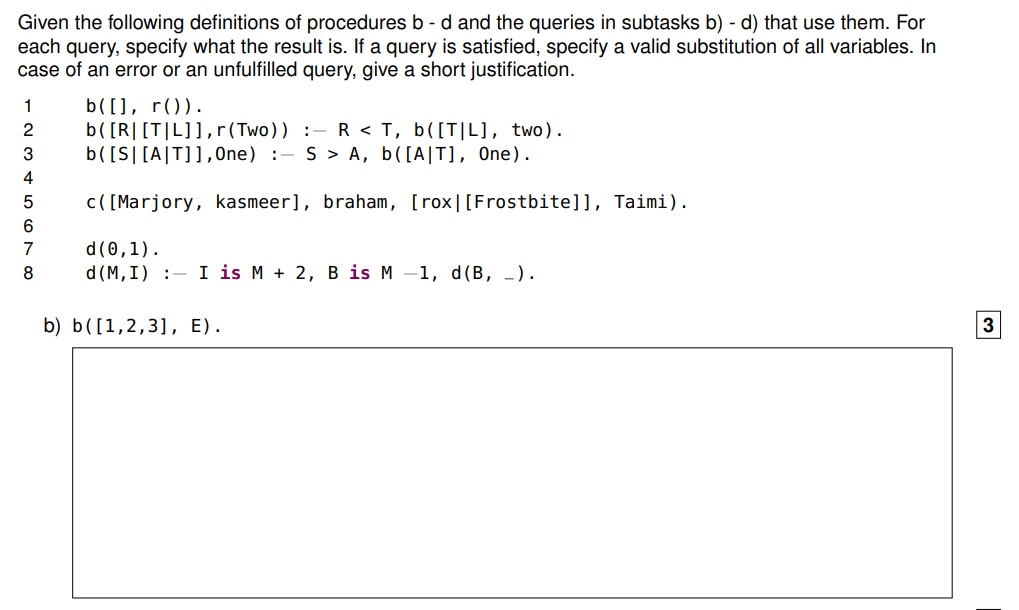
****

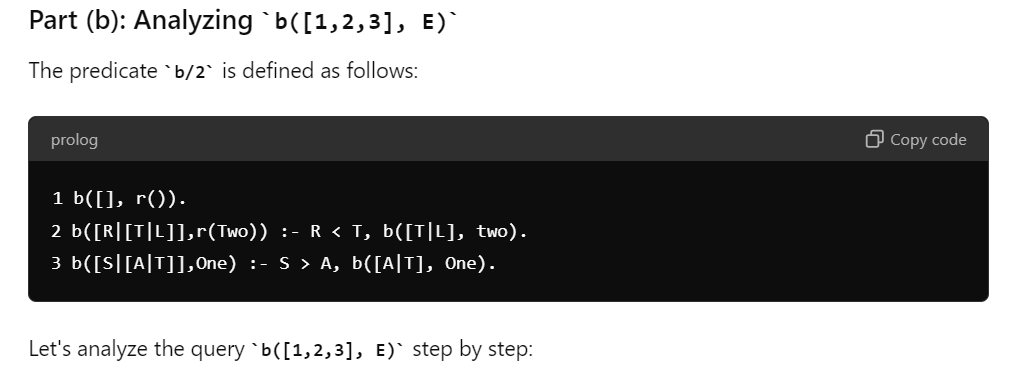
****

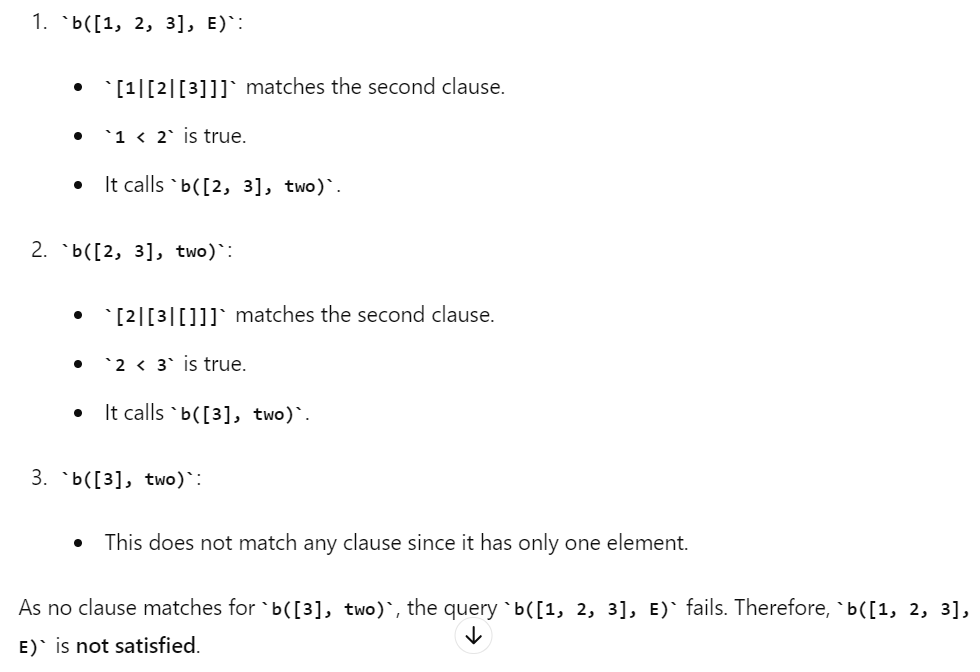
****

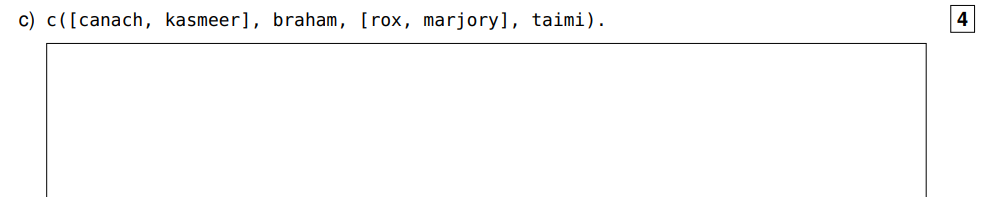
****

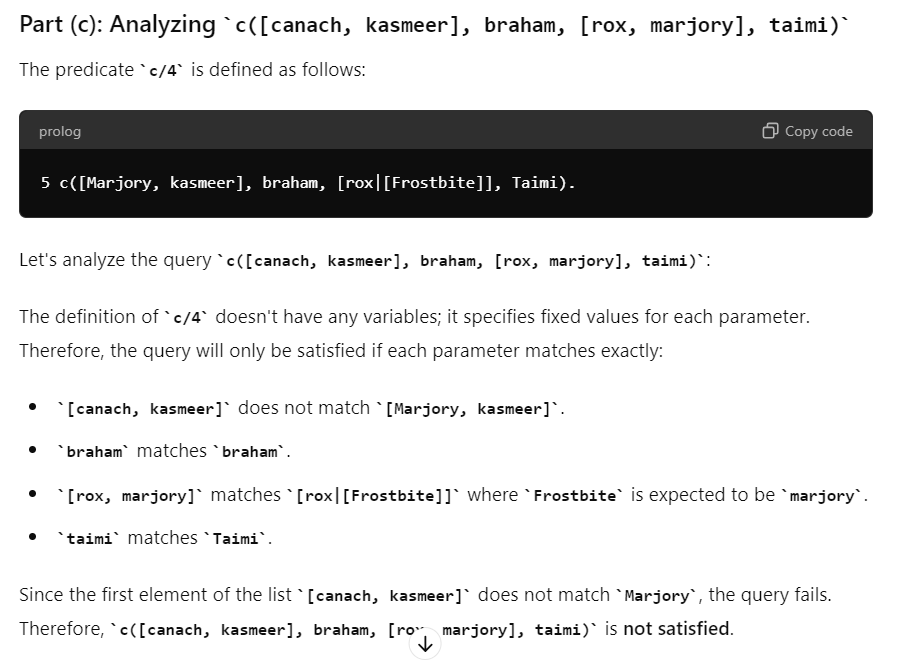
****

****

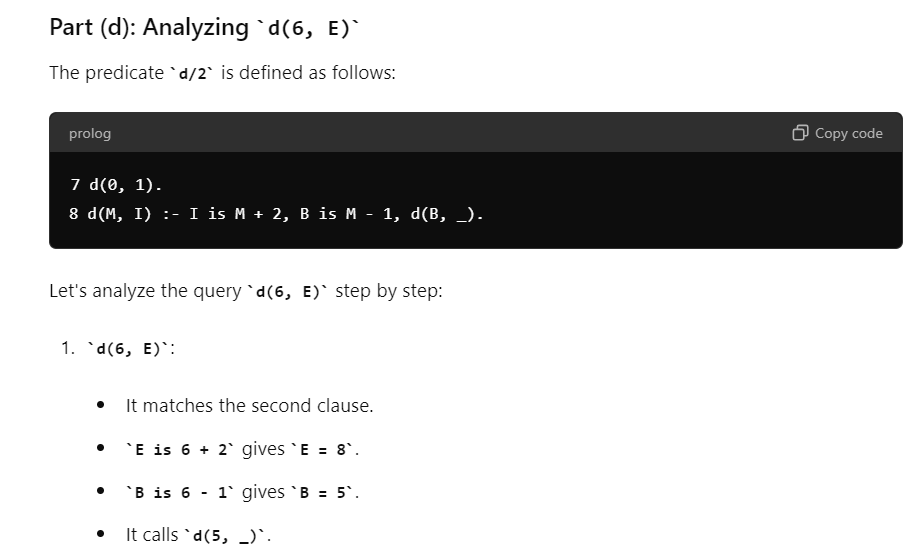
****

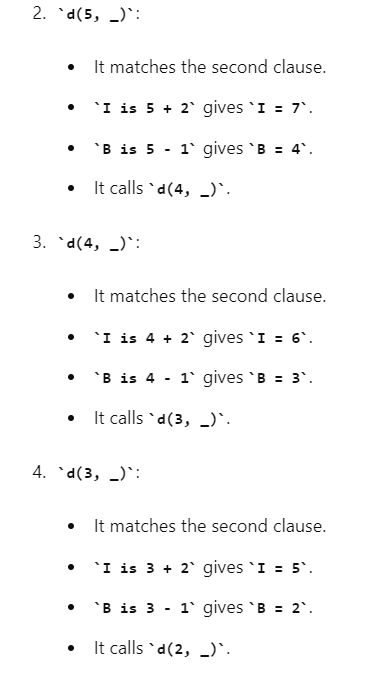
****

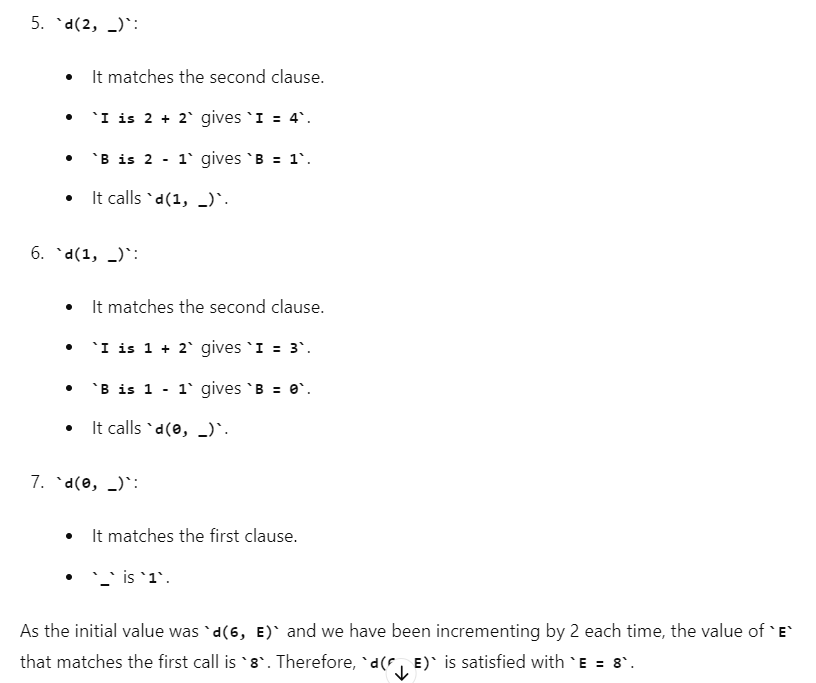
****

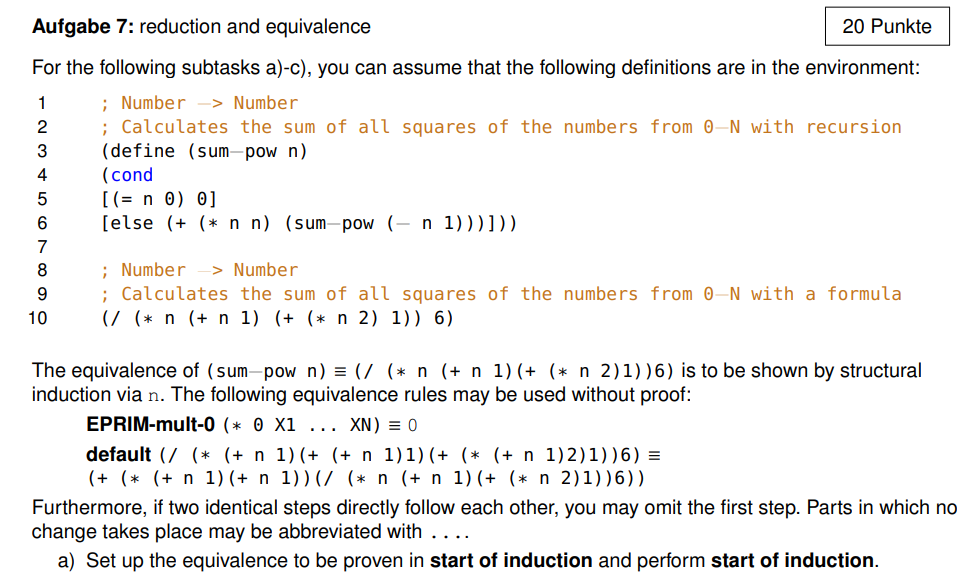
****

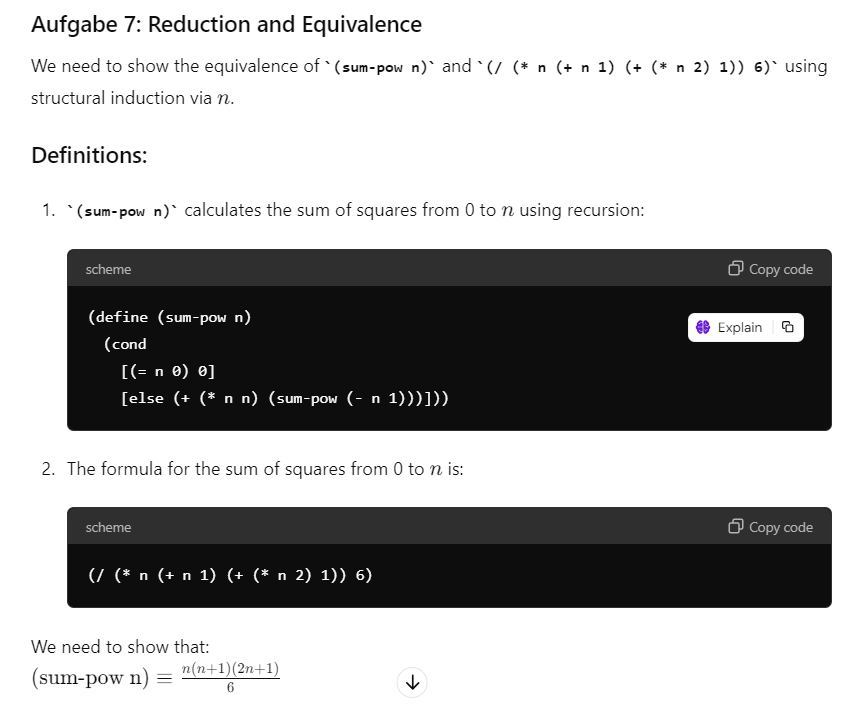
****

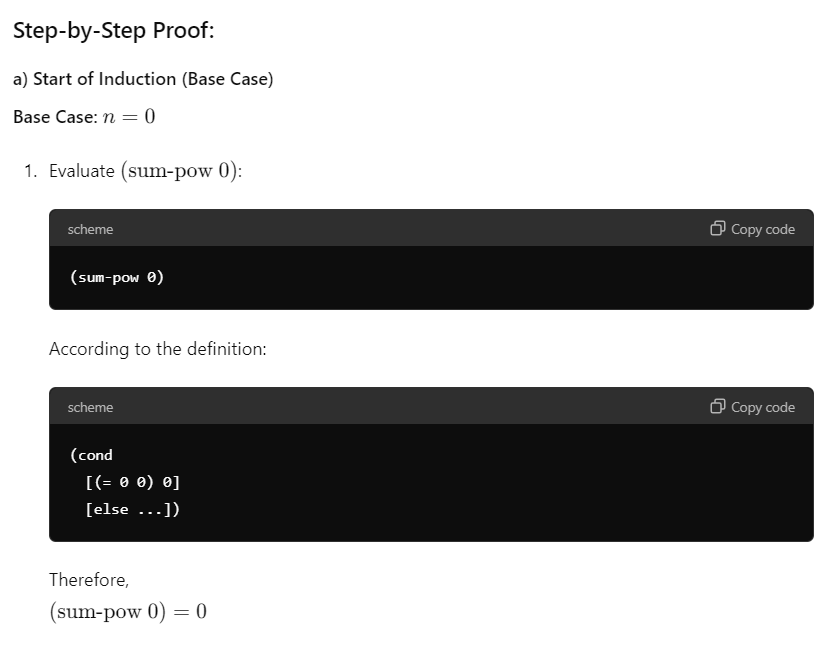
****

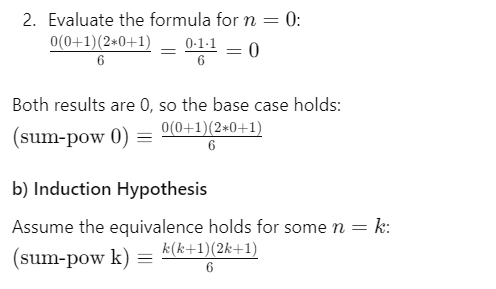
****

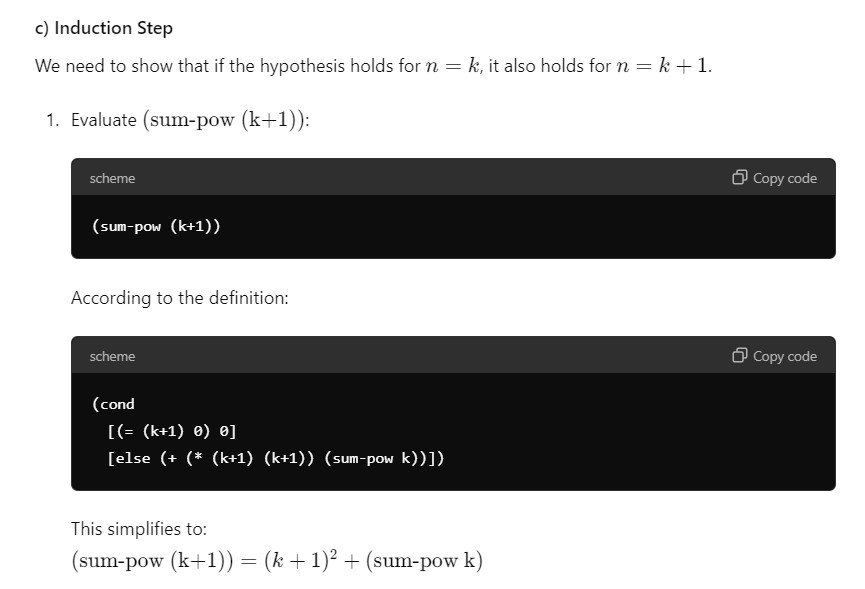
****

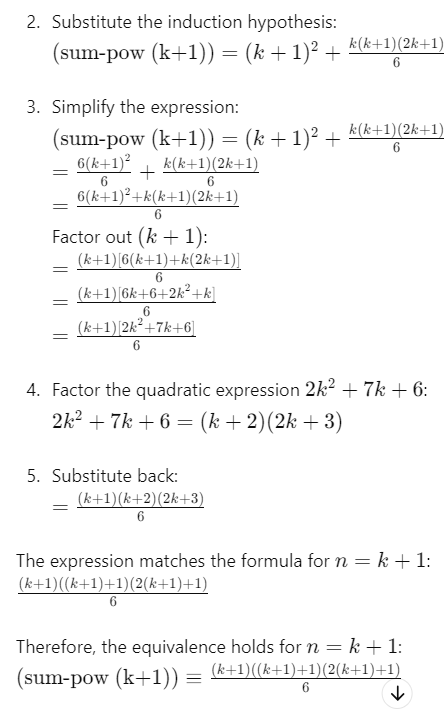
****

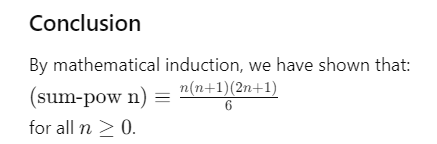
****

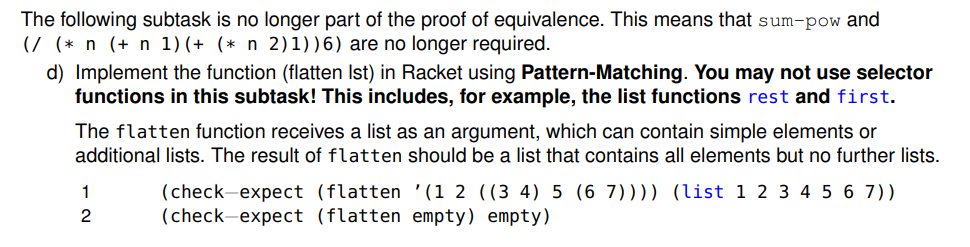
****

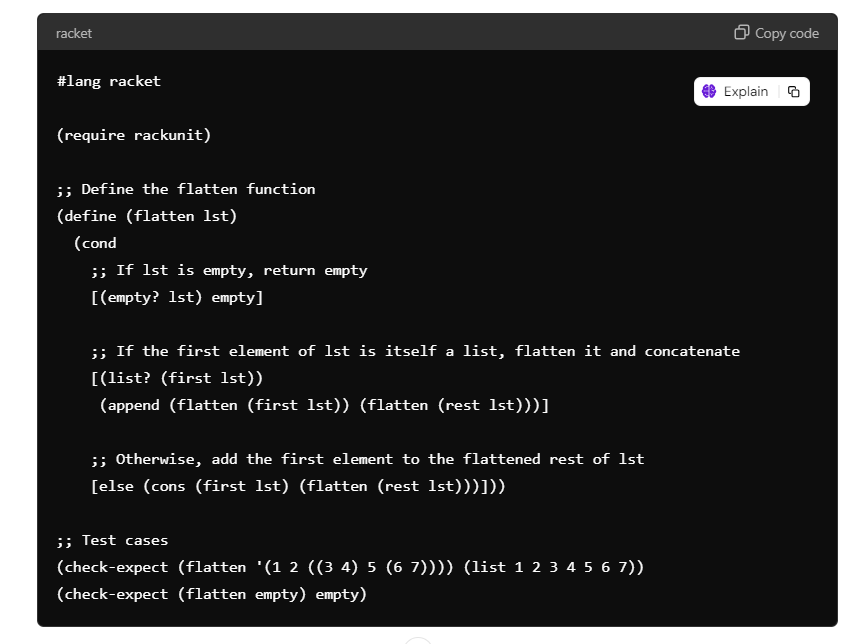
****

****

****

****

****

****

### Explanation

1. **Base Case: [(empty? lst) empty]**
   * Checks if the list lst is empty using empty?. If it is empty, return an empty list.
2. **Recursive Case for Nested Lists: [(list? (first lst)) ...]**
   * Checks if the first element of the list is itself a list using list?.
   * If true, recursively flatten the first element and concatenate it with the flattened rest of the list using append.
3. **Recursive Case for Non-Lists: [else (cons (first lst) (flatten (rest lst)))]**
   * If the first element is not a list, construct a new list with this element and recursively flatten the rest of the list.

### Notes:

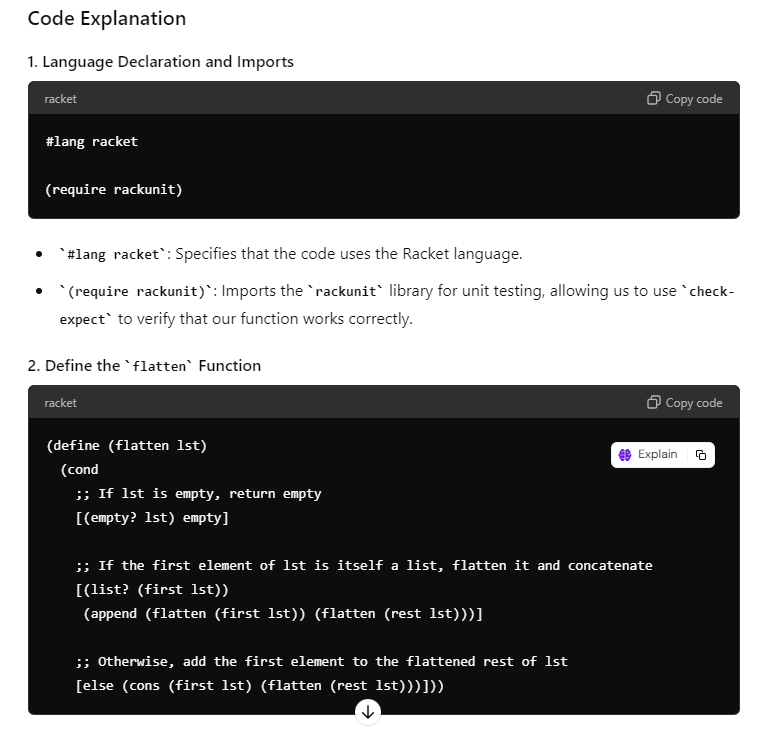
* **list?**: This function checks if an element is a list. It is used here to determine if the current head of the list is a nested list or an atomic value.
* **empty?**: This function checks if the list is empty.

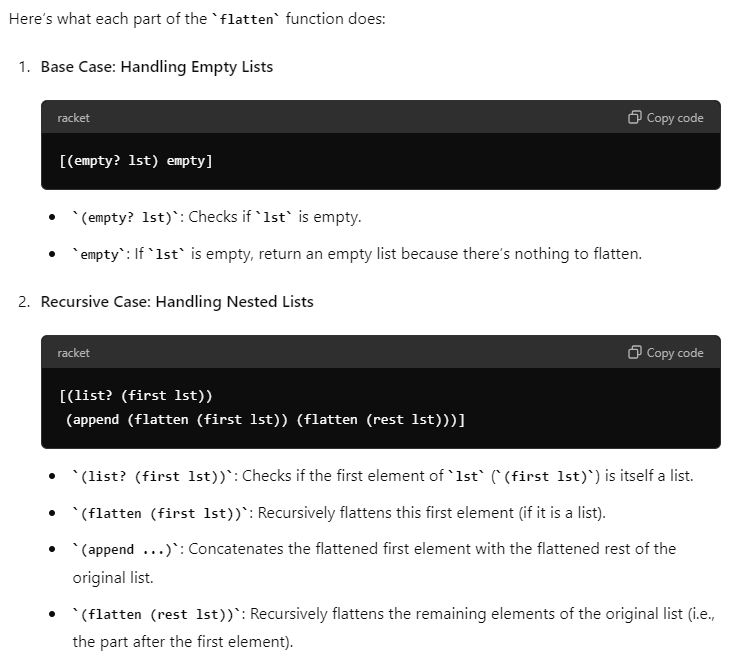
This implementation should work within the constraints of the Intermediate Student with lambda language environment, avoiding the use of functions like pair? that aren't available in this language subset.

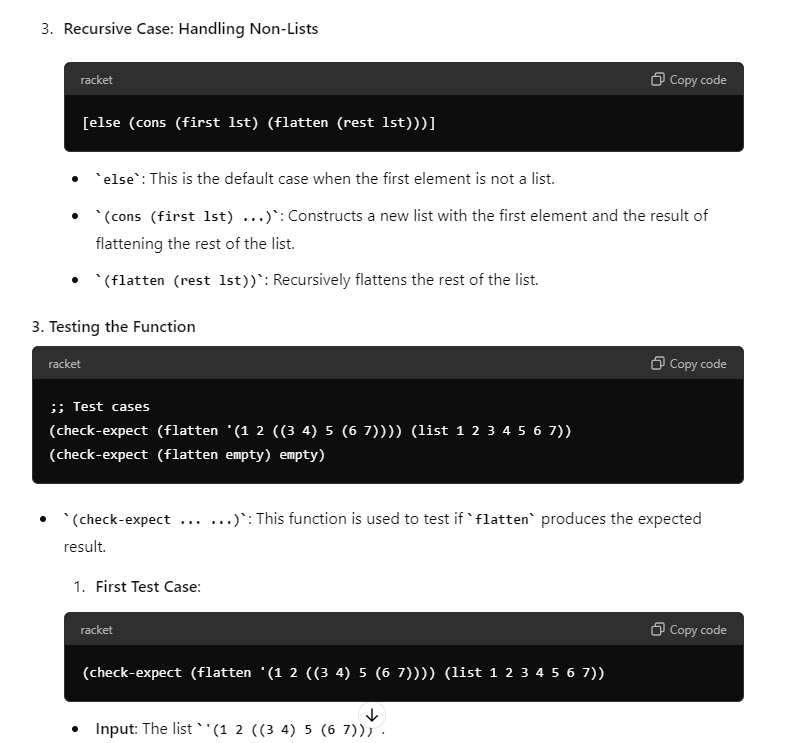
Certainly! Let’s break down the flatten function step by step to understand how it works.

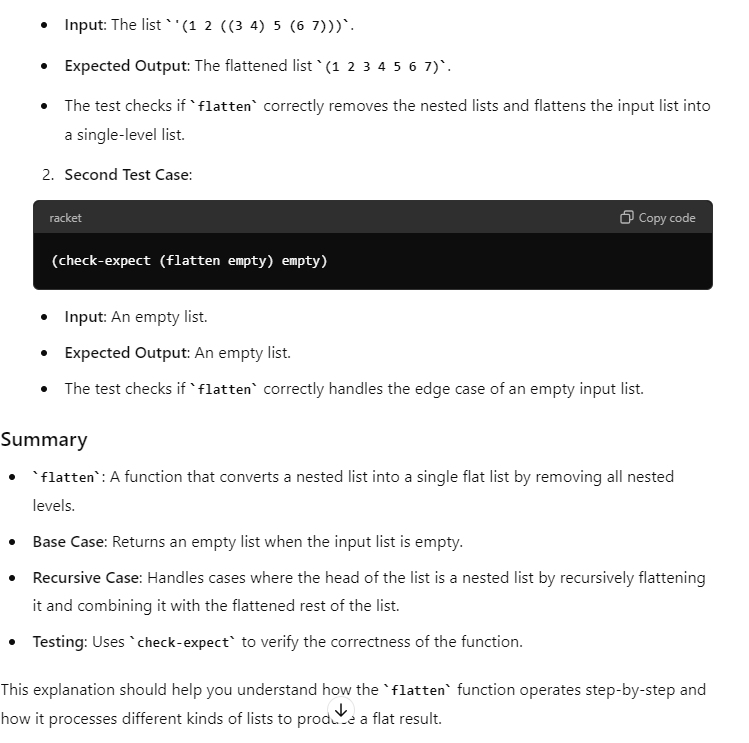
### What is flatten?

The flatten function takes a list that can contain other lists within it and returns a single list with all the elements, but without any nested lists. For example, given a list like '(1 2 ((3 4) 5 (6 7))), the flatten function will produce (1 2 3 4 5 6 7) by removing all nested levels.

****

****

****

****