***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, conditions are checked sequentially from top to bottom, so order matters to ensure correct logic. Place specific conditions before general ones and include a default case (e.g., else) at the end.

**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:

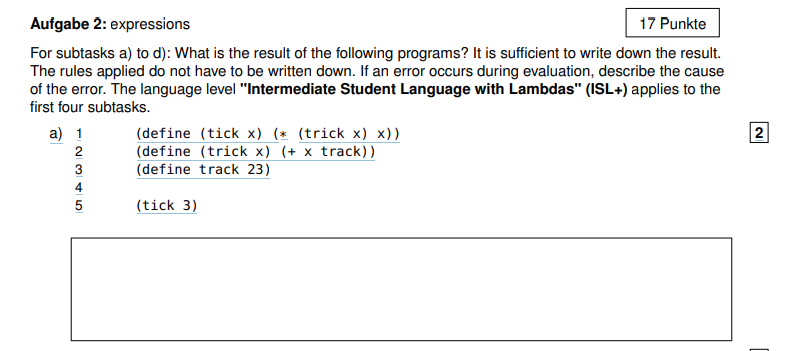
* The function add1 adds 1 to its argument x using the traditional define syntax.
* The function add1-sugar achieves the same result using syntactic sugar with a let binding, where result is bound to the value of (+ x 1).

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:**

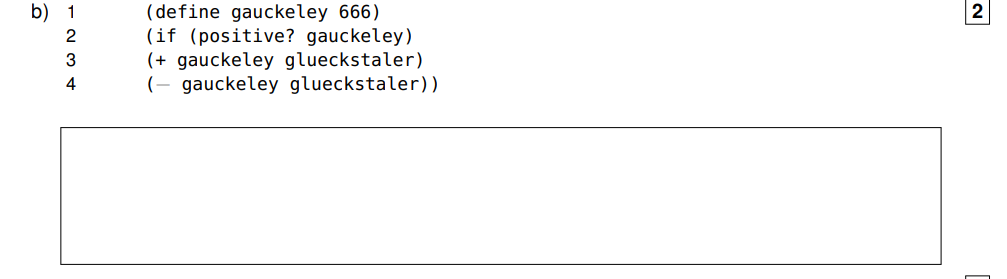
****

**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:**

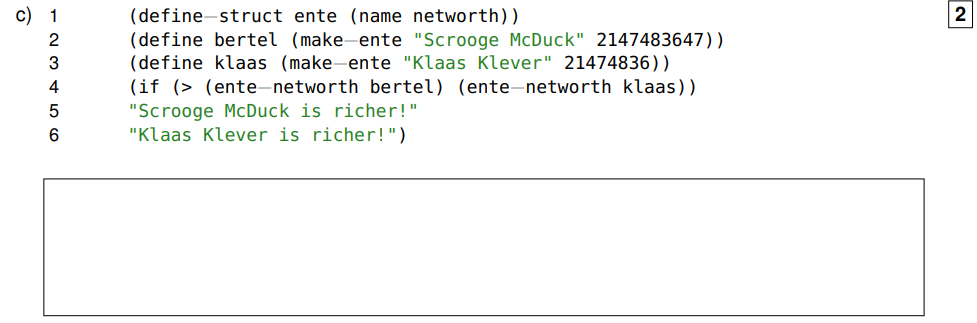
Result: The result depends on the value of glueckstaler, which is not provided in the given code snippet. The expression will evaluate to either 666 + glueckstaler or 666 - glueckstaler, depending on whether gauckeley is positive or not.

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.

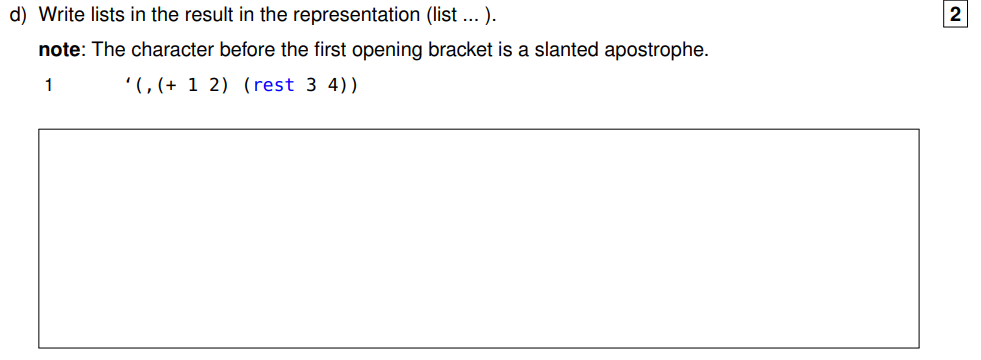
 The if statement compares the networth of bertel and klaas.

 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!".

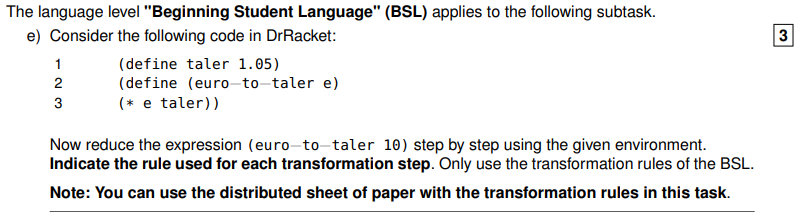
****

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

**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).

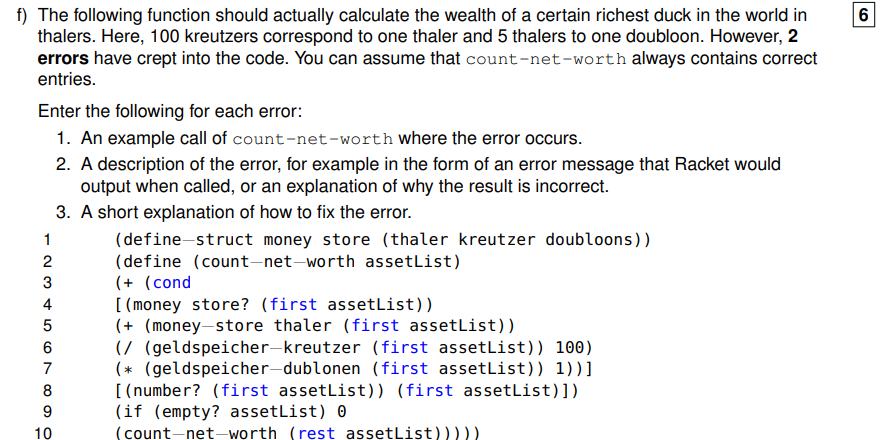
So, the result in the representation (list ...) is (list 3 (list 'rest 3 4)).

****

**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)

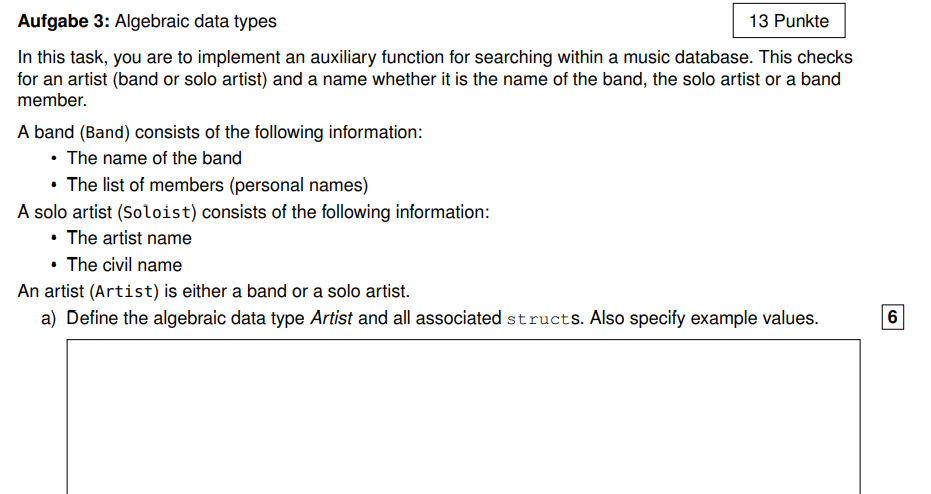
(\* (money-store-doubloons (first assetList)) 5))]

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

(count−net−worth (rest assetList)))))

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

**Q3:**

****

**;; Define the Band struct**

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

**;; Define the Soloist struct**

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

**;; Define the Artist data type using a union**

**(define-struct artist (variant))**

**;; Example values**

**(define coldplay (make-band "Coldplay" (list "Chris Martin" "Jonny Buckland" "Guy Berryman" "Will Champion")))**

**;; Create an artist using the artist struct**

**(define coldplay-artist (make-artist coldplay))**

**;; Example values (after make-soloist definition)**

**(define beyonce (make-soloist "Beyoncé" "Beyoncé Giselle Knowles-Carter"))**

**;; Create another artist using the artist struct**

**(define beyonce-artist (make-artist beyonce))**

**;; Function to check if a name matches the artist or band members**

**(define (search-artist name artist)**

**(cond**

**[(band? (artist-variant artist))**

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

**(member name (band-members (artist-variant artist))))]**

**[(soloist? (artist-variant artist))**

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

**[else #f]))**

**;; Example usage**

**(search-artist "Coldplay" coldplay-artist) ; Output: #t**

**(search-artist "Chris Martin" coldplay-artist) ; Output: #t**

**(search-artist "Beyoncé" coldplay-artist) ; Output: #f**

**(search-artist "Beyoncé" beyonce-artist) ; Output: #t**

**Explanation:**

1. **Struct Definitions**:
   * define-struct band: Defines a struct named band with fields name and members.
   * define-struct soloist: Defines a struct named soloist with fields artist-name and civil-name.
   * define-struct artist: Defines a union type artist which can hold either a band or a soloist.
2. **Example Values**:
   * coldplay: Creates an instance of band representing the band "Coldplay" with specified members.
   * beyonce: Creates an instance of soloist representing the solo artist "Beyoncé".
3. **Creating Artists**:
   * coldplay-artist: Creates an instance of artist holding the coldplay band.
   * beyonce-artist: Creates an instance of artist holding the beyonce soloist.
4. **Function search-artist**:
   * Checks if name matches the band-name or any band-members if artist is a band.
   * Checks if name matches the soloist-artist-name if artist is a soloist.
5. **Example Usage**:
   * Evaluates the search-artist function calls directly. In BSL, the results of these expressions will be displayed in the interactions pane of DrRacket.