Philipps-Universität Marburg
Department of Mathematics and
Computer Science
AG Programming languages and tools

Summer semester 2023
17.07.2023

Prof. Dr. Christoph Bockisch
MSc. Steffen Dick

Lecture exam

Declarative programming

Important notes:

- If you have not already done so, switch off your cell phone immediately!
- Also switch off all sources of noise that are not medically necessary.
- Now remove all unauthorized objects from the table. Only a pen (no red, green or pencil) and drinks are permitted. Also have your student ID card and your identity card or passport ready.
- Write your name and matriculation number on each sheet. Sheets without a name will not be corrected and will result in 0 points! In particular, fill in the following table in block letters:

First name	
Surname	
Matriculation number	
Study subject	
Aimed degree	

- The processing time is **2** hours.
- Do not use your own paper for notes. There is 1 extra sheet at the end of the exam. You can receive additional sheets on request. Make it clear if you are using additional sheets for solutions and also enter your name and matriculation number there.
- No other aids are permitted. Failure to comply will result in exclusion from the exam.
- Multiple, contradictory solutions to a task are awarded 0 points.
- If you have any questions, please contact the tutors quietly.

Overview of achievable points:

Task	1	2	3	4	5	6	7	Total
Points	14	17	13	11	10	15	20	100

Note: This exam is for digital use only. Space for answers has been removed from this exam.

Task 1: Knowledge questions

14 points

Answer the following questions in 1-2 short sentences!

a) What does the term shadowing mean? **2**

b) What must be observed when evaluating the order of cond expressions? **2**

c) What does structural recursion mean? **2**

d) What does the term atom mean in Prolog? Give an example! **2**

e) What is a scope in programming? **2**

f) Briefly explain the term "accumulator variant". **2**

g) What is syntactic sugar? **2**

Task 2: Expressions

17 points

For subtasks a) to d): What is the result of the following programs? It is sufficient to write down the result. The rules applied do not have to be written down. If an error occurs during evaluation, describe the cause of the error. The language level **"Intermediate Student Language with Lambdas" (ISL+)** applies to the first four subtasks.

a) **2**

```
1 (define (tick x) (*  (trick x) x))
2  (define (trick x) (+ x track))
3  (define track 23)
4
5 (tick 3)
```

b) **2**

```
1 (define gauckeley 666)
2  (if (positive? gauckeley)
3    (+ gauckeley glueckstaler)
4     (- gauckeley glueckstaler))
```

c) **2**

```
1 ( define=struct ente (name networth))
2  (define bertel (make=ente "Dagobert Duck" 2147483647))
3  (define klaas (make=ente "Klaas Klever" 21474836))
4  (if (> (ente=networth bertel) (ente=networth klaas))
5    "Scrooge McDuck is richer!"
6    "Klaas Klever is richer!")
```

d) In the result, write lists in the display (list ...). **2**

Note: The character before the first opening bracket is a slanted apostrophe.

```
1 '(,(+ 1 2) (remainder 3 4))
```

The **"Beginning Student Language" (BSL)** level applies to the following subtask.

e) Consider the following code in DrRacket: **3**

```
1  (define taler 1.05)
2  (define (euro=to=taler e)
3    (*  e taler))
```

Now reduce the expression `(euro=to=taler 10)` step by step using the given environment. **Specify the rule used for each transformation step.** Only use the transformation rules of the BSL.

Hint: You can use the sheet of paper with the transformation rules in this exercise.

f) The following function should actually calculate the wealth of a certain richest duck in the world in thalers. Here, 100 kreutzers correspond to one thaler and 5 thalers to one doubloon. However, **2 errors** have crept into the code. You can assume that `count-net-worth` always contains correct entries. **6**

Enter the following for each error:

1. An example call of `count-net-worth` where the error occurs.
2. A description of the error, for example in the form of an error message that Racket would output when called, or an explanation of why the result is incorrect.
3. A brief explanation of how to rectify the error.

```
1  (define=struct money store (taler kreutzer dublonen))
2  (define (count=net=worth assetList)
3    (+ (cond
4      [(moneystore? (first assetList))
5       (+ (moneystore=taler (first assetList))
6         (/ (geldspeicher=kreutzer (first assetList)) 100)
7         (*  (geldspeicher=dublonen (first assetList)) 1))]
```

```
8       [(number? (first assetList)) (first assetList)])
9        (if (empty? assetList) 0
10      (count=net=worth (rest assetList)))))
```

Task 3: Algebraic data types | 13 points

In this task, you are to implement an auxiliary function for searching within a music database. This checks for an artist (band or solo artist) and a name whether it is the name of the band, the solo artist or a band member.

A `band` consists of the following information:

- The name of the band
- The list of members (personal names)

A solo artist (`soloist`) consists of the following information:

- The stage name
- The civil name

An `artist` is either a band or a solo artist.

a) Define the algebraic data type *Artist* and all associated `structs`. Also enter example values. **6**

b) Implement a function `(matches artist name)` that returns `true` if the name matches the band name or the name of a band member or, in the case of a soloist, the artist name or the civil name. **Apply the design recipe for functions over algebraic data types.** **7**

Note: Outsource any required functions.

Task 4: Lists and higher-order functions

11 points

***Note*: You may use the following higher-order functions known from the lecture in task a):**

```
1  ; [X] (X Boolean) (listof X) (listof X)
2  ; Returns a list containing all elements from l
3  ; that fulfill the predicate p
4  (filter p l)
5
6  X Y] (X Y) (listof X) (listof Y)
7  ; Maps all elements from l with f and returns
8  ; the list of results.
9  (map f l)
10
11 X Y] (X Y => Y) Y (listof X) => Y
12 ; Combines all elements of the list l by f. The
13 ; empty list is mapped to base, the elements
14 ; are run through from right to left.
15 (foldr f base l)
```

a) Recreate the functionality of `(myFoldl f base l)`. Apart from `append`, only use the **above-mentioned higher-order functions and lambda expressions** in your implementation. **5**

```
1  X Y] (X Y => Y) Y (listof X) => Y
2  ; Combines all elements of the list l by f. The
3  ; empty list is mapped to base, the elements
4  ; are run through from left to right.
5  (check=expect (myFoldl cons empty (list 1 2 3)) (list 3 2 1))
6  (define (myFoldl f base l)
```

b) Implement the function `(myOrmap proc lst)` **without** using **higher order functions**. **6**

```
1  (check=expect (myOrmap positive? '(1 2 a)) true)
2  X] (X => boolean) List=of=X => boolean
3  ; Applies proc to all elements of the list lst and
4  ; combines the partial results with a logical or.
5  (define (myOrmap proc lst)
```

Task 5: Recursion & scheduling

10 points

a) Implement the function `(myFilter proc lst)`, which filters a list `lst` by the predicate `proc`, using an accumulator. The original order of the list should be retained. Define all auxiliary functions only locally within the function. Also specify the accumulator **invariant.** **7**

b) Consider the following function, which adds up the elements of a list, which can also contain lists. **3**

```
1  (define (mySumList lst)
2    (cond
3      [(emtpy? lst) 0]
4      [(number? (first lst)) (+ (first lst) (mySumList (rest lst)))]
5      [else (+ (mySumList (first lst)) (mySumList (rest lst)))]))
```

Explain why the function `mySumList` terminates and also state what type of recursion (*generative*, *accumulative* or *structural*) is involved. Explain your answer in 1-2 short sentences!

Task 6: Prologue

15 points

a) Define the following procedure in Prolog. The use of library procedures from Prolog is not permitted. However, arithmetic operations such as + or - may be used. You may also use `append/3`. The relation `append(L1, L2, E)` links two lists (L1 and L2) to a third (E). Define the required auxiliary procedures yourself. **5**

`is_palindrome(L)`, which is fulfilled if L is a list whose elements result in a palindrome. A palindrome is a word that can be read the same forwards and backwards (e.g. Girafarig or Farigiraf).

Given the following definitions of procedures b - d and the queries in subtasks b) - d) that use them. For each query, state what the result is. If a query is satisfied, specify a valid substitution of all variables. In the event of an error or an unfulfilled query, give a brief explanation.

```
1 b([], r()).
2  b([R|[T|L]],r(Two)) :- R < T, b([T|L], Two).
3  b([S|[A|T]],One) :- S > A, b([A|T], One).
4
5 c([Marjory, kasmeer], braham, [rox|[Frostbite]], Taimi).
6
7 d(0,1).
8 d(M,I) :- I is M + 2, B is M - 1, d(B, _).
```

b) `b([1,2,3], E).` **3**

c) `c([canach, kasmeer], braham, [rox, marjory], taimi).` **4**

d) `d(6, E).` **3**

Task 7: Reduction and equivalence

20 points

For the following subtasks a)-c), you can assume that the following definitions are in the environment:

```
1  ; Number => Number
2  ; Calculates the sum of all squares of the numbers of with recursion
3  (define (sum=pow n)
4    (cond
5      [(= n 0) 0]
6      [else (+ (*  n n) (sum=pow (- n 1)))]))
7
8  ; Number => Number
9  ; Calculates the sum of all squares of the numbers from with a formula
10   (/ (*  n (+ n 1) (+ (*  n 2) 1)) 6)
```

The equivalence of `(sum=pow n)` ≡ `(/ (* n (+ n 1)(+ (* n 2)1))6)` is to be shown by structural induction over `n`. The following equivalence rules may be used without proof:

EPRIM-mult-0 `(* 0 X1 ... XN)` ≡ `0`

E default `(/ (* (+ n 1)(+ (+ n 1)1)(+ (* (+ n 1)2)1))6)` ≡
`(+ (* (+ n 1)(+ n 1))(/ (* n (+ n 1)(+ (* n 2)1))6))`

Furthermore, if two identical steps directly follow each other, you may omit the first step. Parts in which no change takes place may be abbreviated with `...` may be abbreviated.

a) Establish the **equivalence to be proven** in the **induction start** and carry out the **Induction start** through. **5**

b) Set up the **induction acceptance.** **1**

c) Now establish the **equivalence to be proven** in the **induction step** and then carry out the **induction step.** **8**

The following subtask is no longer part of the proof of equivalence. This means that `sum-pow` and `(/ (* n (+ n 1)(+ (* n 2)1))6)` `is` no longer required.

d) Implement the function (flatten lst) in Racket using **pattern matching**. **You may not use any selector functions in this subtask! This includes, for example, the list functions** `rest` **and** `first`**.** **6**

The `flatten` function receives a list as an argument, which can contain simple elements or additional lists. The result of `flatten` should be a list that contains all elements but no further lists.

```
1  (check=expect (flatten '(1 2 ((3 4) 5 (6 7)))) (list 1 2 3 4 5 6 7))
2  (check=expect (flatten empty) empty)
```