

Solution Exam Programming Languages

Date: Friday, 01.06.2018.

Duration: 90 minutes

Material: You are NOT allowed to use any material (e.g., script, exercises including solutions, notes, electronic devices...)

Number of exercises: 7

Total points: 70

Firstname, lastname: _____

Matriculation number: _____

Write your name on each extra page you deliver.

Consecutively number all pages. Total number of extra pages: _____

Exercise 1 (18 Points)

Answer the following questions. Do not write more than 3 sentences. Each question is worth 2 points.

1. Name three programming paradigms/styles and note their unique characteristics.

2. Are two following Haskell `times` functions equivalent? Justify your answer.

```
times x y = x * y
```

```
times (x,y) = x * y
```

3. What is the difference between monomorphic and polymorphic type? Is the following Haskell function monomorphic or polymorphic? Why?

```
sum [] = 0
```

```
sum (x:xs) = x + sum xs
```

4. What is a normal form of a λ expression? How does one reach it? Does the expression $(\lambda x. y) (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)))$ have a normal form? Justify your answer.

5. Is it possible to define a recursive expression in λ calculus? Justify your answer.

6. What is the difference between abstract and concrete syntax?

7. What is static semantics and what is dynamic semantics?

8. Explain what are parametric polymorphism and coercion and provide an example for each of them.

9. How are questions answered in a Prolog program? For example, `mother(charles, M)`.

Answer:

1. **imperative** *program = algorithms + data*

functional *working in the terms of mathematical function, the state of a variable cannot be modified*

logic *a program is decomposed into facts and rules, used to develop new facts, that is express thing that you know and rules for inferring new thing*

OO *good for modeling*

2. *The first function is curried. `times` is a function of one argument that returns a function as its result. The second code examples represents the `times` function of two arguments.*

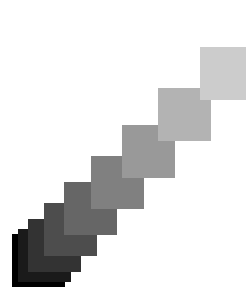
3. *“Monomorphic” means “every value has a single, unique type”. “Polymorphic” means that a value can have more than one type. The function is polymorphic, since it accepts a list whose elements can be any type on which the + function can be applied.*
4. *A λ expression is in normal form if it can no longer be reduced by beta or eta reduction rules. A λ expression can be reduced to its normal form by applying beta or eta reduction rules. Yes, it has the normal form: y , obtained by the application of one eta reduction.*
5. *It is not possible to express recursion in λ calculus, since we cannot bind a name to “global function”, but we can only work with anonymous λ s.*
6. *Concrete syntax is needed for parsing. Abstract syntax suffices for semantics specifications.*
7. *The static semantics tells us which (syntactically valid) programs are semantically valid (i.e., which are type correct) and the dynamic semantics tells us how to interpret the meaning of valid programs.*
8. *Questions in Prolog are answered by:*
 - *matching goals against facts or rules (resolution),*
 - *unifying variables with terms (unification), and*
 - *backtracking when subgoals fail.*

Exercise 2 (6 Points)

A developer wants to write a program in PostScript which draws squares as in [Figure 1a](#):



(a) Desired output



(b) Actual output

All the squares have the same dimensions, the distances between lower left corners of any two consecutive squares are the same, and each square is one nuance of a lighter grey color than the previous one, going from black to white colour. But the actual output is as in [Figure 1b](#). Her code is as following:

```
/squares {  
  /n exch def  
  /size exch def  
  /cy exch def  
  /cx exch def  
  0 1 n {  
    /i exch def  
    cx i 10 mul add  
    cy i 10 mul add  
    size  
    i n div  
    square  
  } for  
} def  
/square{  
  /grayScale exch def  
  /size exch def  
  /halfSize size 2 div def  
  /cy exch def  
  /cx exch def  
  cx halfSize sub cy halfSize sub moveto  
  cx halfSize sub cy halfSize add lineto  
  cx halfSize add cy halfSize add lineto  
  cx halfSize add cy halfSize sub lineto  
  closepath  
  grayScale setgray  
  fill  
} def  
% usage cx cy size n squares  
200 300 100 10 squares
```

Explain what is the error. Correct the code to produce the desired output.

Answer:

The problem is that variables `cx` and `cy` get redefined on the stack, since there is no defined dictionary for any of procedures. The correct code is:

```
/squaresDict 8 dict def % defined dictionary for the squares procedure
/squareDict 10 dict def % defined dictionary for the square procedure
/squares {
    squaresDict begin % dictionary for the squares procedure
    /n exch def
    /size exch def
    /cy exch def
    /cx exch def
    0 1 n {
        /i exch def
        cx i 10 mul add
        cy i 10 mul add
        size
        i n div
        square
    } for
    end % end dictionary
} def
/square{
    squareDict begin % dictionary for the square procedure
    /grayScale exch def
    /size exch def
    /halfSize size 2 div def
    /cy exch def
    /cx exch def
    cx halfSize sub cy halfSize sub moveto
    cx halfSize sub cy halfSize add lineto
    cx halfSize add cy halfSize add lineto
    cx halfSize add cy halfSize sub lineto
    closepath
    grayScale setgray
    fill
    end %end dictionary
} def
% usage cx cy size n squares
200 300 100 10 squares
```

Exercise 3 (13 Points)

NB: For this exercise, you are allowed to use only arithmetical built-in Haskell functions!

- (4 points) Write a Haskell function `magic n` which returns the magic number of the argument `n`. The magic number of a natural number `n` is equal to the product of its digits.

- (4 points) Write a Haskell function `number s` which returns the number represented by the elements of the list `s`. Consider that elements of the list are only natural numbers smaller than 10. For example, `number [1, 2, 3]` will return as the result the number 123.

- (5 points) Infer the type of the following function and explain your steps. Is the function monomorphic or polymorphic? Explain.

```
apply f g [] = []
apply f g (x:xs)
  | f x < 0 = (g x) : apply f g xs
  | otherwise = (f x) : apply f g xs
```

Answer:

- `magic n`
| `n < 10 = n`
| `otherwise = (n `mod` 10) * magic (n `div` 10)`
- `len [] = 0`
`len (x:xs) = 1 + len xs`

`number [] = 0`
`number [x] = x`
`number (x:xs) = x * 10 ^ (len xs) + number xs`
- `apply ::`
a -> b -> c -> d since apply takes three arguments and returns something
a -> b -> c -> [e] since d is of type list
(f -> g) -> b -> c -> [e] since f takes one argument
(Ord h => f -> h) -> b -> c -> [e] since < :: Ord a => a -> a -> Bool
(Ord h => c -> h) -> b -> c -> [e] since f takes x as an argument
(Ord h => c -> h) -> (i -> j) -> c -> [e] since g takes one argument
(Ord h => c -> h) -> (c -> j) -> c -> [e] since g takes x as an argument
(Ord h => c -> h) -> (c -> e) -> c -> [e] since g x is part of the resulting list
list
(Ord e => c -> e) -> (c -> e) -> c -> [e] since f x is part of the resulting list
list

The result is:

```
:type apply
apply :: (Ord a => b -> a) -> (b -> a) -> b -> [a]
The function is polymorphic, since both c and e can be distinct types.
```

Exercise 4 (3 Points)

Consider the following λ -expressions. Indicate which occurrences of variables are bound and which ones are free in the expressions.

- (1 point)

$(\lambda x . x y) (\lambda x y z . x) x z$

- (2 points)

$((\lambda x . \lambda y z . z y) (\lambda x y z . y x) y) z (\lambda x z . z x) (\lambda x . z x)$

Answer:

b = bound

f = free

- $(\lambda x . x y) (\lambda x y z . x) x z$

b	f	b	f	f

- $((\lambda x \lambda y \lambda z . z y) (\lambda x \lambda y \lambda z . y x) y) z (\lambda x \lambda z . z x) (\lambda x . z x)$

b	b	b	b	f	f	f

Exercise 5 (12 Points)

We represent non-negative integers with the following Lambda expressions:

$$\begin{aligned} 0 &\equiv \lambda f . \lambda x . x \\ 1 &\equiv \lambda f . \lambda x . fx \\ 2 &\equiv \lambda f . \lambda x . f(fx) \\ &\vdots \\ n &\equiv \lambda f . \lambda x . f^n x \end{aligned}$$

Suppose you have defined the function **if** and the operations **times**, **pred** and **isOne**. Consider the following recursive (and hence not valid) definition for the factorial calculation:

$$\mathbf{fact} = \lambda n . \mathbf{if} (\mathbf{isOne} \ n) \ \mathbf{1} \ (\mathbf{times} \ n \ (\mathbf{fact} \ (\mathbf{pred} \ n)))$$

To do:

- (4 points) Translate the **fact** definition into a proper definition, i.e., using the Y combinator.

- (8 points) Write down the reduction sequence to demonstrate that factorial of 3 is 6.

Answer:

If we abstract the name **fact**, we get : $t = \lambda f . \lambda n . \mathbf{if} (\mathbf{isOne} \ n) \ \mathbf{1} \ (\mathbf{times} \ n \ (f \ (\mathbf{pred} \ n)))$.

$(Y \ t)$ is a non-recursive equivalent of the above **fact** definition.

$(Y \ t) \ 3 \equiv$ /* Fixpoint Theorem tells us that $Y \ t = t \ (Y \ t)$ */

$t \ (Y \ t) \ 3 \equiv$

$(\lambda f . \lambda n . \mathbf{if} (\mathbf{isOne} \ n) \ \mathbf{1} \ (\mathbf{times} \ n \ (f \ (\mathbf{pred} \ n)))) (Y \ t) \ 3 \equiv$ /* $f = Y \ t$, $n = 3$. */

$\mathbf{if} (\mathbf{isOne} \ 3) \ \mathbf{1} \ (\mathbf{times} \ 3 \ ((Y \ t) \ (\mathbf{pred} \ 3))) \equiv$ /* $\mathbf{isOne} \ 3 = \mathbf{False}$ */

$\mathbf{times} \ 3 \ ((Y \ t) \ 2) \equiv$

$\mathbf{times} \ 3 \ (t \ (Y \ t) \ 2) \equiv$

$\mathbf{times} \ 3 \ ((\lambda f . \lambda n . \mathbf{if} (\mathbf{isOne} \ n) \ \mathbf{1} \ (\mathbf{times} \ n \ (f \ (\mathbf{pred} \ n)))) (Y \ t) \ 2) \equiv$

$\mathbf{times} \ 3 \ (\mathbf{if} (\mathbf{isOne} \ 2) \ \mathbf{1} \ (\mathbf{times} \ 2 \ ((Y \ t) \ (\mathbf{pred} \ 2)))) \equiv$

times 3 (**times** 2 ((Y t) 1)) ≡
times 3 (**times** 2 (t (Y t) 1)) ≡
times 3 (**times** 2 ((λ f. λ n. **if** (**isOne** n) **1** (**times** n (f (**pred** n))))(Y t) 1)) ≡
times 3 (**times** 2 (**if** (**isOne** 1) **1** (**times** 1 ((Y t) (**pred** 1)))))) ≡
times 3 (**times** 2 1) ≡
times 3 2 ≡
6

3. (3 points) jerry is sick and no longer eats, instead he responds with "I am sick", while the others still respond with "munch". Update the code correspondingly.

Answer:

1. *animal*

2. `animal.sleep=function () {return this.name + " sleeps"}`

3. `jerry.eat=function () {return "I am sick!!!"}`

Exercise 7 (10 Points)

Create a finite collection of definite clause grammar rules to check whether a sentence is grammatically correct. A sentence can be composed of the following words: A sentence can be composed of the following words:

article a, the

noun girl, girls.

verb play, plays.

A sentence must be in the form subject-predicate.

- subject is formed out of an article and a noun. For example, a girl.
- predicate is a verb

The sentence should be grammatically correct in a sense that the article a cannot be used in front of a noun in plural. If a subject is in plural, the following verb must be play, and if the subject is in singular, the following verb should be plays.

Write a Prolog question to produce all correct sentences in the grammar.

You can test your program with the following examples:

```
a girl plays // True
a girl play // False
the girls plays // False
the girls play // True
girls plays // False
girls play // False
a girls play // False
```

Answer:

```
sentence(Number) --> noun_phrase(Number), verb_phrase(Number).
noun_phrase(Number) --> determiner(Number), noun(Number).
verb_phrase(Number) --> verb(Number).
```

```
determiner(singular) --> [a].
determiner(_) --> [the].
determiner(plural) --> [].
noun(singular) --> [girl].
noun(plural) --> [girls].
verb(singular) --> [likes].
verb(plural) --> [like].
```


Points

Exercise 1

Task	Points	Score
1	2	
2	2	
3	2	
4	2	
5	2	
6	2	
7	2	
8	2	
9	2	
Total	18	

Exercise 2

Task	Points	Score
1	6	
Total	6	

Exercise 3

Task	Points	Score
1	4	
2	4	
3	5	
Total	13	

Exercise 4

Task	Points	Score
1	1	
2	2	
Total	3	

Exercise 5

Task	Points	Score
1	4	
2	8	
Total	12	

Exercise 6

Task	Points	Score
1	2	
2	3	
3	3	
Total	8	

Exercise 7

Task	Points	Score
1	10	
Total	10	

TOTAL

Exercise	Points	Score
1	18	
2	6	
3	13	
4	3	
5	12	
6	8	
7	10	
Total	70	