ALFP lecture: Summary

January 2020

An exam is scheduled on February 3 (Monday), from 17:00, in room 045C

- The exam will be a written test from the material presented. Below, you can find some typical questions for the written test.
- The final grade will be computed as the average mean (*media aritmetică*) between
  - your grade for the labworks
  - your grade at the exam (written test)

1 General questions

1. Indicate the main programming paradigms from software engineering and their characteristic features.
2. Express the following sentences in First Order Logic.
   (a) Every man is mortal.
   (b) Nobody walked on Mars.
   (c) Every man has a father and a mother.
   (d) Somebody took the umbrella and closed the door.
   (e) Somebody took all apples from the table.
   (f) Felix is a smart but lazy cat.
   (g) John likes all singing birds but he does not like dogs.
3. Formalize the following sentences as formulas in First Order Logic:
   (a) Every natural number has a successor.
   (b) Nothing is better than taking a nap.
   (c) There is no such thing as negative integers.
   (d) Everybody likes babies.
   (e) Logic plays an important role in all areas of computer science.
   (f) The renter of a car pays the deductible in case of an accident.

2 Logic Programming

1. What is Logic Programming?
2. Andrew, Ann, and Adam are siblings and so are Bill, Beth and Basil. Describe the relationships between these persons using as few clauses as possible.
3. Suppose the following predicates have been already defined:
   - `father(F,X)` to express the fact that `F` is father of `X`
   - `mother(M,X)` to express the fact that `M` is mother of `X`
   - `male(X)` to express the fact that `X` is male
   - `female(X)` to express the fact that `X` is female

   Define the following predicate symbols (with obvious intended interpretations):

   ```
   grandchild(X,Y)
   sister(X,Y)
   brother(X,Y)
   cousins(X,Y)
   uncle(X,Y)
   aunt(X,Y)
   ```
4. An ancestor is a parent, a grandparent, a great-grandparent etc. Define a relation `ancestor/2` which is to hold if someone is an ancestor of somebody else.
5. (Operations on lists) Define the following predicates on lists:

   `append(P,S,L)` which holds if `L` is the list produced by appending `P` and `S`
   `prefix(P,L)` which holds if list `P` is prefix of list `L`
   `suffix(S,L)` which holds if list `S` is suffix of list `L`
   `sublist(M,L)` which holds if `M` is sublist of list `L`
   `last(L,X)` which holds if `L` is a non-empty list whose last element is `X`.
6. Suppose `A` and `B` are list of numbers in strictly increasing order. Define the following relations:
   (a) `merge1(A,B,L)` which holds if `L` contains all all elements of `A` and `B` in increasing order. Duplicate elements should occur only once in `L`.
   (b) `merge2(A,B,L)` which holds if `L` contains all all elements of `A` and `B` in increasing order. Duplicate elements should occur only twice in `L`.
   (c) `intersect(A,B,L)` which holds if `L` is the list of elements which occur both in `A` and `B`, in increasing order.
   (d) `dropSmaller(A,N,L)` which holds if list `L` is obtained from list `A` by removing all elements smaller than number `N`.
   (e) `drop(A,N,L)` which holds if list `L` is obtained from list `A` by removing the first `N` elements.

7. An informal way to define a sorted list of numbers is:

   - The empty list is a sorted list of numbers.
   - A list of length 1 is a sorted list of numbers if its element is a number.
   - A list of length ≥ 2 is a sorted list of numbers if its first element is $\leq$ than its second element, and its tail is a sorted list of numbers.

   Formalize this definition with Prolog rules and facts for the tail-recursive predicate `isSorted(L)` which holds if `L` is a sorted list of numbers.

8. A nested list is either the empty list, or a list whose elements are symbols or nested lists. For example, the following are nested lists:

   ```
   []    [a [x [a [] [b]]] [y z]]    [[[x]]]    [[] [[b]] a]
   ```

   (a) Define the predicate `nestedList(L)` which holds iff `L` is a nested list.

   (b) The flattened form of a nested list `L` is the list of symbols in `L`. For example, the flattened forms of the previous lists are the are nested lists:

   ```
   []    [a x a b y z]    [x]    [b a]
   ```

   Define the predicate `flatten(L,F)` which holds if `F` is the flattened form of `L`.

   Note: You can use the predefined predicate `atom(X)` which holds if `X` is a symbol.

9. Consider the predicates defined by

   ```
   insert(X,L,[X|L]).
   insert(X,[H|T],[H|R]):-insert(X,T,R).

   mistery([],[]).
   mistery([H|T],R):-mistery(T,T1),insert(H,T1,R).
   ```

   (a) How many answers has the query `insert(1,[2,3,4],R)`?

   (b) What is the relation between `L` and `R` if `mistery(L,R)` holds?

   (c) Suppose `sortedList(L)` is a predefined predicate which returns `true` iff `L` is a sorted list of numbers. Use the predicates `isSorted` and `mistery` to define the predicate

   `sortedList(L,S)`

   which holds if `S` is the sorted version of the list of numbers `L`.

10. Write a tail-recursive definition of the predicate `power6(N)` which holds iff `N` is a power of 6. You can make use of the arithmetic operator

    ```
    X is A mod B
    ```

    which binds `X` to the remainder of dividing integer `A` by integer `B`.

11. Write a tail-recursive definition of the predicate `power23(N)` which holds iff `N` is a number of the form $2^a \cdot 3^b$ where a, b are non-negative integers.

12. Write a tail-recursive definition of the predicate `evenLength(L)` which holds iff `L` is a list whose length is even.

13. Assume given the following collection of facts which indicate existing roads between towns:

```
road(timisoara,arad).
road(arad,curtici).
road(timisoara,lugoj).
road(lugoj,deva).
```

The roads ar bidirectional (if there is a road from X to Y, there is also a road from Y to X). Define a predicate `route(X,Y,T)` which holds if `T` is a list that represents a route from `X` to `Y`. For example

```
?-route(arad,deva,T).
T = [arad, timisoara, lugoj, deva]
true

?-route(curtici, timisoara, [curtici, arad, timisoara]).
true
```

14. Consider the following Prolog program

```
% facts for the friendship relation
friend(radu, elena).
friend(elena, mihai).
friend(zoe, viorel).
```

(a) Define, using the cut-fail combination, the relation `notFriend(X,Y)` which holds when we can not deduce that `X` and `Y` are friends.

(b) What are the answers to the queries:

i. `?-notFriend(radu, elena).`

ii. `?-notFriend(geo, zoe).`

15. Formalize the following sentences as rules and facts in Prolog.

All sportsmen are powerful. Everyone who is intelligent and powerful will succeed in life. Rick is a sportsman. Alex knows Rick. All sportsmen who know Alex and Rick are smart.

3 Functional Programming

1. Indicate the main features of functional programming.

2. Consider the function defined by

```
(define (mistery m)
  (if (and (integer? m) (>= m 0))
     (if (< m 10)
         m
         (mistery (quotient m 10))
      'undefined))
```

   (a) Indicate the values of the following function calls:

```
(mistery "I am Sam")
(mistery -3)
(mistery 916)
(mistery 6)
(mistery 1435)
```

   (b) In general, what is the value of `(mistery` m`)` when m is a positive integer?

   Note that `(quotient m n)` takes as inputs two integers `m` and `n` > 0, and returns the remainder of division of `m` by `n`.

3. Consider the function $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined by

$$f(a,b) = \begin{cases} 0 & \text{if } b = 0, \\ f(2\,a, b/2) & \text{if } b > 0 \text{ is even}, \\ a + f(2\,a, (b-1)/2) & \text{if } b > 0 \text{ is odd}. \end{cases}$$

   (a) What is the value of f for (i) $a = 3, b = 4$, (ii) $a = 4, b = 3$, and (iii) a, b two arbitrary natural numbers?

   (b) Write a recursive definition in Racket for f.

   (c) Write a tail recursive definition in Racket for f.

4. Consider the function $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ defined by

$$g(a,b) = \begin{cases} 1 & \text{if } b = 0, \\ g(a^2, b/2) & \text{if } b > 0 \text{ is even}, \\ a \cdot g(a^2, (b-1)/2) & \text{if } b > 0 \text{ is odd}. \end{cases}$$

   (a) What is the value of g for (i) $a = 3, b = 4$, (ii) $a = 4, b = 3$, and (iii) a, b two arbitrary natural numbers?

   (b) Write a recursive definition in Racket for g.

   (c) Write a tail recursive definition in Racket for g.

5. Consider the function `makelist` : $\mathbb{N} \to \mathbb{N}$ defined by

```
(define (makelist n)
   (if (= n 0)
      '()
      (cons n (makelist (- n 1)))))
```

   (a) What are the values of `(makelist 5)` and `(length (makelist 5))`?

   (b) The predefined function call `(even? n)` returns `#t` if `n` is an even inteer, and `#f` otherwise. What do the following expressions compute?

   b1)
```
(map (lambda x) (* x x)
             (filter (lambda (y) (even? y)) (makelist 6)))
```

   b2)
```
(let ([lst (makelist 6)])
        (/ (apply + lst) (length lst)))
```

   (c) In general, what are the values of `(makelist n)` and `(length (makelist n))` when `n` $\in \mathbb{N}$?

6. Consider the function `my-list` defined by

```
(define (my-list n)
  (if (= n 0)
     null
     (cons 1 (map (lambda (x) (+ x 1)) (my-list (- n 1))))))
```

   (a) What is the value of `(my-list 3)`?

   (b) In general, what is the value of `(my-list` n`)` when n is a positive integer?

7. Define recursively the function `(c-frac L)` which takes as input a nonempty list `L` of non-negative integers $c_1, c_2, \ldots, c_n$ and computes the value of the fraction

$$c_1 + \cfrac{1}{c_2 + \cfrac{1}{c_3 + \cfrac{1}{\ddots + \cfrac{1}{c_n}}}}$$

More about tail recursion

1. Write a tail-recursive definition of the function `reverse(L)` which reverses a list `L`.

2. Write a tail-recursive definition of the function `fib(n)` which computes the value of the n-th Fibonacci number f_n. Remember that Fibonacci numbers are defined as follows:

$$f_1 = f_2 = 1, \quad f_n = f_{n-1} + f_{n-2} \quad \text{for all } n \geq 2.$$

3. Write tail-recursive definitions for the following functions

   (a) $fact(n)$ which computes the factorial $1 \cdot 2 \cdot \ldots \cdot n$ of a natural number n.

   (b) $gcd(m, n)$ which computes the greatest common divisor of non-negative integers a and b.

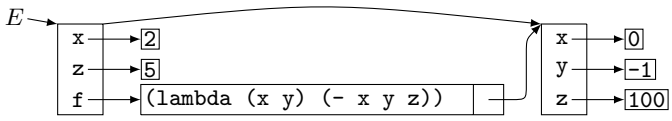
   (c) $expt(x, n)$ which computes the value of x^n for some non-negative integer n.

let, let*, letrec

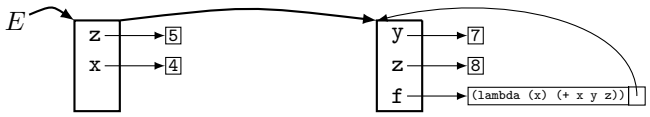
1. What are the values of the following expressions?

   a)
```
(let ([x 1]
      [y 2]
      [z 3])
  (let ([y x]
        [z y])
    (list x y z)))
```
   b)
```
(let ([x 1]
      [y 2]
      [z 3])
  (let* ([y x]
         [z y])
    (list x y z)))
```

2. What is the value of `(* (f y x) z)` in the environment E depicted below?



3. Let E be the environment



   (a) What are the values of `z` and `(f z)` in environment E?

   (b) Draw the environment which is obtained by evaluating in E the function definition

```
(define f (lambda (x) (+ x y z))
```

   (c) What is the value of `(f z)` in E after we evaluate the previous function definition in E?

4. Suppose $P_1P_2 \cdots P_n$ is a polygon, and every vertex P_i has coordinates (x_i, y_i) with x_i, y_i real numbers. Define the following functions:

(a) `(dist P Q)` which takes as inputs the vertices

$$\texttt{P=(cons } x_1\ y_1\texttt{)} \text{ and } \texttt{Q=(cons } x_2\ y_2\texttt{)},$$

and returns the distance between them. Note that the distance between `P` and `Q` is $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$.
For example, `(dist '(1 . 2) '(4 . 6))` should return `5` because $\sqrt{(4-1)^2+(6-2)^2}=5$.

(b) `(perim L)` which takes as input the list

`L = '((`x_1 `. `y_1`) (`x_2 `. `y_2`) ... (`x_n `. `y_n`) (`x_1 `. `y_1`))`

of coordinates of vertices $P_1, P_2, \ldots, P_n, P_1$ and computes the perimeter of the polygon $P_1P_2\cdots P_n$.
Define the function `perim` recursively, and make the function `dist` local to the body of `perim`. For example:

```
> (perim '((0 . 0) (3 . 0) (0 . 4) (0 . 0)))
24
```

5. Use Euclid's algorithm to write a tail recursive definition of the function `(gcd m n)` which computes the greatest common divisor of two positive integers `m`, `n`.

6. A positive integer is a Hamming number if it belongs to the set $\{2^a3^b5^c \mid a, b, c \in \mathbb{N}\}$ where $\mathbb{N} = \{0, 1, 2, \ldots\}$.

   (a) Write a recursive definition of the function `(hamming? n)` which takes as input a positive integer `n` and returns: `#t` if `n` is a Hamming number, and `#f` otherwise.

   (b) Let `n` be a Hamming number. What is the result of the function call `(get3 n)` if the function `get3` is defined by

```
(define (get3 n)
   (define (get3acc n a b c)
     (cond [(= n 1) (list a b c)]
           [(= (remainder n 2) 0) (get3acc (/ n 2) (+ a 1) b c)]
           [(= (remainder n 3) 0) (get3acc (/ n 3) a (+ b 1) c)]
           [(= (remainder n 5) 0) (get3acc (/ n 5) a b (+ c 1))]))
   (get3acc n 0 0 0))
```

7. Define, using tail recursion, the function `(ones `n`)` which returns the list made of n elements, all equal to `1`. For example

```
> (ones 4)      > (ones 0)      > (ones 2)
'(1 1 1 1)      > '()           '(1 1)
```

8. Consider the function `foo` defined by

```
(define (foo lst)
  (if (null? lst)
      '(())
      (cons null
            (map (lambda (x) (cons (car lst) x))
                 (foo (cdr lst))))))
```

(a) Is the definition of `foo` recursive? Is the definition of `foo` tail recursive?

(b) What is the result of the function call `(foo '(a b))`?

(c) In general, what is the result of the function call `(foo lst)` when `lst` is a list?

9. Define recursively the function `(merge lst1 lst2)` which takes as inputs two lists of numbers sorted in increasing order, and returns the list of all elements in `lst1` and `lst2` in increasing order. For example

```
> (merge '(1 3 4 6) '(2 5 7 8))
'(1 2 3 4 5 6 7 8)
```

10. Define recursively the function `(bitnumber?` n`)` which returns `#t` if n is a non-negative integer whose digits are only 0 or 1. For example

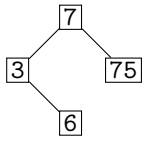
```
> (bitnumber? 0)    > (bitnumber 1001101)    > (bitnumber 1203)
#t                  #t                       #f
```

Datatypes

1. Consider binary trees of integers represented as follows

   ⟨bt⟩ ::= 'leaf | (list ⟨integer⟩ ⟨bt⟩ ⟨bt⟩)

   For example, the binary tree

   has the representation `'(7 (3 leaf (6 leaf leaf)) (75 leaf leaf))`.

   A binary search tree is a binary tree such that, every subtree `(list k T1 T2)` satisfies the condition: the integers in subtree `T1` are smaller than `k`, and the integers in subtree `T2` are larger than `k`.

   Define the following functions:

   (a) `(bt? T)` which returns `#t` if `T` is a search tree, and `#f` otherwise.

(b) `(insert n T)` which returns the binary search tree produced by inserting a node with key `n` $\in \mathbb{N}$ in the binary search tree `T`.

(c) `(max-key T)` which returns the maximum key of a node in tree `T`.

2. Consider the string of numbers which satisfy the following recursive relation:

$$s_0 = 1, s_1 = 2, s_2 = 4, \quad s_n = s_{n-1} - 2 \cdot s_{n-2} + s_{n-3} \text{ if } n > 2.$$

Write a tail recursive definition of the function `(s` n`)` which computes the value of s_n.

3. Write a tail recursive definition of the function defined by

$$f : \mathbb{N} \to \mathbb{N}, f(n) = \begin{cases} 1 & \text{if } n = 0, \\ 3 \cdot f(n/2) & \text{if } n > 0 \text{ is even}, \\ 1 + f(n-1) & \text{if } n > 0 \text{ is odd}. \end{cases}$$

SUGGESTION: Define, using tail-recursion, the function `(f-acc` $n\ P\ S$`)` which computes $P \cdot f(n) + S$. Note that the value of $f(n)$ coincides with the value of `(f-acc` n `1 0)`.

4. Write a tail recursive definition of the function `(reverse-append L1 L2)` which returns the reverse of append of lists `L1`, `L2`. For example:

```
> (reverse-append '(1 2) '(a b))    > (reverse-append '(a b c) '())
'(b a 2 1)                          '(c b a)
```

5. The Cartesian product `L1` $\times$ `L2` of two lists can be defined recursively as follows:

- If either `L1` or `L2` is the empty list, then `L1` $\times$ `L2` is the empty list.
- Otherwise, `L1` is a nonempty list with a first element `a`, and tail `Ls`, and `L2` is a list of elements $\mathtt{b}_1, \ldots, \mathtt{b}_m$. In this case, $L_1 \times L_2$ can be computed as follows:
  - ▶ First, we compute the list `L` of pairs (`list` (`cons a` $\mathtt{b}_1$) ... (`cons a` $\mathtt{b}_m$)). We can use map to compute this ist.
  - ▶ Next, we append `L` with the result of the Cartesian product $Ls \times L_2$.

Use the previous informal definition to write a recursive definition of the function `(cp L1 L2)` for the computation of the Cartesian product of `L1` and `L2`. For example:

```
> (cp '(a b) '(1 2 3))
'((a . 1) (a . 2) (a . 3) (b . 1) (b . 2) (b . 3))
```

6. Let `lst` be a list of binary trees of integers represented as follows:

⟨`bt`⟩ `::= 'lf` | ⟨`integer`⟩ | `(list` ⟨`integer`⟩ ⟨`bt`⟩ ⟨`bt`⟩`)`

Write a tail recursive definition of the function

`(int-list lst)`

which returns the list of all integers encountered in the binary trees from `lst`, as we traverse them in inorder, from left to right. For example

```
> (int-list '((2 lf 81) (3 (1 lf (7 6 lf)) (5 12 lf))))
'(2 81 1 6 7 3 12 5)
```

7. A nested list of numbers is defined by the grammar:

   ⟨Nlist⟩ ::= ⟨number⟩ | (list ⟨Nlist⟩ ... ⟨Nlist⟩)

   (a) Define the boolean function `(Nlist? L)` which holds if and only if `L` is a nested list of numbers.

   (b) Define the function `(sumN L)` which computes the sum of numbers that occur in the nested list of numbers `L`.

map, filter, foldl, length, ...

You should know what the built-in functions `apply`, `map`,`filter`, `foldl`, `foldr`, and `length` are doing.

1. Indicate the results of the function calls:

   (a) `(filter even? (map (lambda (x) (+ x 1)) '(2 9 7 1 4 3 8)))`

   (b) `(foldl cons null '(a b c d e))`

   (c) `(foldr (lambda (x y) (cons x (cons x y))) null '(a b c d))`

   (d)
   ```
   (map (lambda (l) (- (apply max l) (apply min l)))
        '((1 2) (9 2 3) (-7 4 5)))
   ```

2. Use `map`, `filter`, and `apply` to define the following functions:

   (a) `(f1 flst v)` which takes as input a list of functions `flst=(list` f_1 f_2 ... f_n`)` and returns the value of $\frac{f_1(v) + f_2(v) + \ldots + f_n(v)}{n}$.

   For example, `(f1 (list sin cos (lambda (x) (+ x 1))) 5)` returns the value of $\frac{\sin(5) + \cos(5) + 6}{3}$

   (b) `(f2 L)` which takes as input a list `L` of numbers and symbols, and returns the pair `'(`p_1 . p_2`)` where p_1 is the percentage of numbers which occur in `L`, and p_2 is the percentage of symbols which occur in `L`.

   For example:

   ```
   > (f2 '(a 1 4.5 -7 b 4 0 abc -1 6))
   '(70 . 30)
   ```

   because 70% are numbers and 30% are symbols.

3. Consider nested lists defined by the grammar

⟨nlist⟩ ::= null | (cons ⟨symbol⟩ ⟨nlist⟩)|(cons ⟨nlist⟩ ⟨nlist⟩)

and the following function definition which expects as input a nested list `nlst` ∈ ⟨nlist⟩:

```
(define (bar nlst [acc null])
  (cond [(null? nlst) acc]
        [(symbol? (car nlst)) (bar (cdr nlst) `(,@acc ,(car nlst)))]
        [#t (bar `(,@(car nlst) ,@(cdr nlst)) acc)]))
```

(a) Is the definition of `foo` tail recursive? Explain your answer.

(b) What is the value of the function call `(bar '(x (a (() b ((c))) (d (e)))))`?

4. Consider the structures

```
(struct rect (a b) #:transparent)
(struct rtTriangle (a b) #:transparent)
(struct circle (r) #:transparent)
(struct (square l) #: transparent)
```

They are used to represent geometric shapes: `(rect a b)` is a rectangle with edges of lengths `a` and `b`, `(rtTriangle a b)` is a right triangle with catheti of lengths `a` and `b`, `(circle r)` is a circle with radius of length `r`, and `(square l)` is a square with edge of length `l`. Define the following functions:

(a) `(area shape)` which takes as input an instance of the previous four structures and computes its area.

(b) `(perimeter shape)` which takes as input an instance of the previous four structures and computes its perimeter.

(c) `(squaresArea lst)` which takes as input a list of geometric shapes and computes the sum of areas of the squares in `lst`.

Streams

You should know how to use nullary functions to define some useful streams, and to define some useful operations on streams.

1. Suppose `s1` is a stream for the infinite list

(list a_1 a_2 a_3 a_4 ...)

and `s2` is a stream for the infinite list

(list b_1 b_2 b_3 b_4 ...)

Define the following operations:

(a) `(s-duplicate s1)` which returns the stream for the infinite list

`(list` a_1 a_1 a_2 a_2 a_3 a_3 a_4 a_4 ...`)`

(b) `(s-interleave s1 s2)` which returns the stream for the infinite list

`(list` a_1 b_1 a_2 b_2 a_3 b_3 a_4 b_4 ...`)`

2. Suppose `s1` is a stream for an infinite list of numbers in increasing order

   `(list` a_1 a_2 a_3 a_4 ...`)`

   and `s2` is also stream for an infinite list of numbers in increasing order

   `(list` b_1 b_2 b_3 b_4 ...`)`

   Define the operation

   ```
   > (s-xor s1 s2)
   ```

   which returns the stream for the elements, in increasing order, that occur in only one of the streams `s1`, `s2`.

3. Use the functions `s-map`, `s-filter`, and `s-add` from the lecture notes to define the stream for the infinite list of numbers

   `(list` s_1 s_2 s_3 s_4 s_5 ...`)`

   where $s_1 = 1, s_2 = 3$, and $s_n = 4 \cdot s_{n-1} - 2 \cdot s_{n-2}$ if $n > 2$.

4 Lecture 12: Macros

1. What are the values of `(c1 1)` and `(c2 1)` after evaluating the following piece of code?

   ```
   (define-values (c1 c2 x) (values 1 1 4))

   (let ([x 1]
         [y x])
     (* y (call/cc (lambda (k) (set! c1 k) x))))

   (let* ([x 1]
          [y x])
     (+ x (call/cc (lambda (k) (set! c2 k) y))))
   ```

2. Consider the macro identifier `foo` defined by

```
(define-syntax foo
  (syntax-rules ()
    [(foo) null]
    [(foo a) `(a)]
    [(foo a as ...) '(as ... a)]))
```

Indicate the values of the following expressions:

(a) `(foo 1 (+ 2 3) 3)`

(b) `(foo (+ 2 3))`

A Answers to selected exercises

2. (a) Every man is mortal.
   $\forall X.man(X) \rightarrow mortal(X)$.

   (b) No man ever walked on Mars.
   $\neg\exists X.(man(X) \wedge walked(X, mars))$.

   (c) Every man has a father and a mother.
   $\forall X.(man(X) \rightarrow (\exists F.father(F, X) \wedge \exists M.mother(M, X)))$.

   (d) Somebody took the umbrella and closed the door.
   $\exists X.took(X, umbrella) \wedge closed(X, door)$.

   (e) Somebody took all apples from the table.
   $\exists X.\forall Y.(apple(Y) \wedge onTable(Y) \rightarrow took(X, Y))$.

   (f) Felix is a smart but lazy cat.
   $cat(felix) \wedge smart(felix) \wedge lazy(felix)$.

   (g) John likes all singing birds but he does not like dogs.
   $(\forall X.bird(X) \wedge sings(X) \rightarrow likes(john, X)) \wedge$
   $(\forall X.dog(X) \rightarrow \neg likes(john, X))$.

3. (a) Every natural number has a successor.
   $\forall n.natural(n) \rightarrow \exists m.successor(n, m)$.

   (b) "Nothing is better than taking a nap" can be rephrased as follows: "If something (x) is good then a nap is better (than x)".
   $\forall x.good(x) \rightarrow better(nap, x)$.

   (c) "There is no such thing as negative integers" can be rephrased as follows: "Every integer x is not negative".
   $\forall x.integer(x) \rightarrow \neg negative(x)$.

   (d) Everybody likes babies.
   $\forall x.likes(x, babies)$.

   (e) "Logic plays an important role in all areas of computer science" can be rephrased "If x is an area of computer science then logic plays an important role in x."
   $\forall x.computerScienceArea(x) \rightarrow importantRule(x, logic)$.

   (f) "The renter of a car pays the deductible in case of an accident" can be rephrased "If x is a car and y is the renter of x and y makes an accident, then y pays the deductible".
   $\forall x.\forall y.car(x) \wedge renter(y, x) \wedge hasAccident(y) \rightarrow paysDeductible(y)$.

B Logic Programming

4.
```
parent(P,X):-father(P,X).
parent(P,X):-mother(P,X).
ancestor(A,X):-parent(A,X).
ancestor(A,X):-parent(A1,X),ancestor(A,A1).
```

5.
```
append([],T,T).
append([H|T],L,[H|R]):-append(T,L,R).

prefix(P,L):-append(P,_,L).
suffix(S,L):-append(_,S,L).
sublist(S,L):-prefix(P,L),suffix(S,P).
last(L,X):-suffix([X],L).
```

6.
```
merge1([],L,L) :- !.
merge1(L,[],L) :- !.
merge1([H|A],[H|B],[H|L]) :- !,merge1(A,B,L).
merge1([H1|A],[H2|B],[H1|L]) :- H1<H2,!,merge(A,[H2|B],L).
merge1(A,[H|B],[H|L] :- merge(A,B,L).

dropSmaller([],_,[]):-!.
dropSmaller([H|T],N,L):-H<N,!,dropSmaller(T,N,L).
dropSmaller(L,_,L).

drop(A,0,A) :- !.
drop([],_,[]) :- !.
drop([H|T],N,L) :- N1 is N-1, drop(T,N1,L).
```

7.
```
isSorted([]).
isSorted([X]):-integer(X).
isSorted([X,Y|T]) :-
       integer(X),
       integer(Y),
       X =< Y,
       isSorted([Y|T]).
```

8. (a)
```
nestedList([]).
nestedList([H|T]):-atom(H),!,nestedList(T).
nestedList([H|T]):-nestedList(H),!,nestedList(T).
```

   (b)
```
flatten(L,R) :- flattenAcc(L,[],A) ,reverse(A,R).

flattenAcc([],A,A).
flattenAcc([H|T],A,R):-atom(H),!,flattenAcc(T,[H|A],R).
flattenAcc([ [] | T],A,R):- !,flattenAcc(T,A,R).
flattenAcc([ [H|T1] | T],A,R):- flattenAcc([H,T1 | T],A,R).
```

10
```
power6(1).
power6(N):- R is N mod 6, R == 0, Q is N / 6, power6(Q).
```

11.
```
power23(1):-!.
power23(N) :- R is N mod 2, R==0, !, Q is N / 2, power23(Q).
power23(N) :- R is N mod 3, R==0, Q is N / 3, power23(Q).
```

12.
```
evenLength([]).
evenLength([_,_|T]) :- evenLength(T).
```

13.
```
route(X,Y,T):-routeAcc(X,Y,[Y],T).
routeAcc(X,X,T,T).
routeAcc(X,Y,A,T):-road(Z,Y),notMember(Z,A),routeAcc(X,Z,[Z|A],T).
routeAcc(X,Y,A,T):-road(Y,Z),notMember(Z,A),routeAcc(X,Z,[Z|A],T).

% notMember(X,L) holds if X is not member of list L
notMember(_,[]):-!.
notMember(X,[X|_]):-!,fail.
notMember(X,[_|T]):-notMember(X,T).
```

C Functional Programming

3. (a) $f(a, b)$ computes the value of $a \cdot b$.

   (b)
```
(define (f a b)
      (cond [(= b 0) 0]
            [(even? b) (f (* 2 a) (/ b 2))]
            [(odd? b) ((+ a (f (* 2 a) (/ (- b 1) 2))))]
      ))
```

   (c)
```
(define (f a b)
       ; (f-acc a b s) computes s+a*b
       (define (f-acc a b s)
          (cond [(= b 0) s]
                [(even? b) (f-acc (* 2 a) (/ b 2) s)]
                [(odd? b) (f-acc (* 2 a) (/ (- b 1) 2) (+ s a))]))
       (f-acc a b 0))
```

4. (a) $g(a, b)$ computes the value of a^b.

   (b)
```
(define (g a b)
      (cond [(= b 0) 1]
            [(even? b) (f (* a a) (/ b 2))]
            [(odd? b) ((* a (f (* a a) (/ (- b 1) 2))))]
      ))
```

   (c)
```
(define (g a b)
       ; (g-acc a b p) computes p*a^b
       (define (g-acc a b p)
          (cond [(= b 0) p]
                [(even? b) (g-acc (* a a) (/ b 2) p)]
                [(odd? b) (g-acc (* a a) (/ (- b 1) 2) (* p a))]))
       (g-acc a b 1))
```

7.
```
(define (c-frac L)
    (cond [(= (length L) 1) (car L)]
          [#t (+ (car L) (/ 1 (c-frac (cdr L))))]))
```

More about tail recursion

1. The following definition of `(reverse L)` is tail-recursive because

   `(reverse L)=(reverse-acc L null)`

   and the definition of `(reverse-acc L A)` is tail-recursive.

```
; (reverse L) returns the reverse of list L
(define (reverse L)
   ; (reverse-acc L) returns the result of
   ; concatenating the reverse of L with A
   (define (reverse-acc L A)
       (cond [(null? L) A]
             [#t (reverse-acc (cdr L) (cons (car L) A))]))
   (reverse-acc L null))
```

2.
```
(define (fib n)
   (define (fib-acc n f1 f2)
      (if (= n 2)
          f2
          (fib-acc (- n 1) (fib-acc f2 (+ f1 f2)))))
   (if (= n 1) 1 (fib-acc n 1 1)))
```

3. (a) The following definition of `(fact n)` is tail-recursive because

   `(fact n)=(fact-acc n 1)`

   and the definition of `(fact-acc n a)` is tail-recursive.

```
(define (fact n)
   ; (fact-acc n a) computes n!*a
   (define (fact-acc n a)
      (if (= n 0) a (fact-acc (- n 1) (* n a))))
   (fact-acc n 1))
```

   (b)
```
(define (gcd a b)
    (if (= b 0) a (gcd b (remainder a b))))
```

   (c) The following definition of `(expt n)` is tail-recursive because

   `(expt x n)=(expt-acc x n 1)`

   and the definition of `(expt-acc x n a)` is tail-recursive.

```
(define (expt x n)
   ; (expt-acc x n a) computes x^n · a
   (define (expt-acc x n a)
      (cond [(= n 0) a]
            [(even? n) (expt-acc (* x x) (/ n 2) a)]
            [(odd? n) (expt-acc (* x x) (/ (- n 1) 2) (* x a))]))
   (expt-acc x n 1))
```