Question 1 (10 points, λ-calculus), Variant 1

Recall Church-encoded numerals:

$$\begin{aligned}
c_0 &:= \lambda s.\lambda z.z \\
c_1 &:= \lambda s.\lambda z.s\ z \\
c_2 &:= \lambda s.\lambda z.s\ (s\ z) \\
c_3 &:= \lambda s.\lambda z.s\ (s\ (s\ z)) \\
&\ldots
\end{aligned}$$

1. **(5 points)** Write down a lambda term that represents the function $f(n) = 2^n$. The answer should be a of the form $f := t$, where t is a pure lambda term (without any references to definitions). In particular, instead of c_1 you should explicitly write $\lambda s.\lambda z.s\ z$.
2. **(5 points)** Write down call-by-value evaluation sequence for $f\ c_1$. The answer consist of a series of pure lambda terms (starting with expanded version of $f\ c_1$). Each step should be a single call-by-value reduction.

Solution

1. First, let's express $f(n)$ using two equations:

$$\begin{aligned}
f(0) &= 1 \\
f(n+1) &= f(n) + f(n)
\end{aligned}$$

Recall the sum of Church numerals:

$$n + m := \lambda s.\lambda z.n\ s\ (m\ s\ z)$$

Assuming $r = f(n)$, the sum $f(n) + f(n)$ can be represented as the term

$$\lambda s.\lambda z.r\ s\ (r\ s\ z)$$

Now, we can express f as a lambda term:

$$f := \lambda n.n\ (\lambda r.\lambda s.\lambda z.r\ s\ (r\ s\ z))\ (\lambda s.\lambda z.s\ z)$$

2.

$$\begin{aligned}
&f\ c_1 && (1)\\
&\equiv (\lambda n.n\ (\lambda r.\lambda s.\lambda z.r\ s\ (r\ s\ z))\ (\lambda s.\lambda z.s\ z))\ (\lambda s.\lambda z.s\ z) && (2)\\
&\to (\lambda s.\lambda z.s\ z)\ (\lambda r.\lambda s.\lambda z.r\ s\ (r\ s\ z))\ (\lambda s.\lambda z.s\ z) && (3)\\
&\to (\lambda z.(\lambda r.\lambda s.\lambda z.r\ s\ (r\ s\ z))\ z)\ (\lambda s.\lambda z.s\ z) && (4)\\
&\to (\lambda r.\lambda s.\lambda z.r\ s\ (r\ s\ z))\ (\lambda s.\lambda z.s\ z) && (5)\\
&\to \lambda s.\lambda z.(\lambda s.\lambda z.s\ z)\ s\ ((\lambda s.\lambda z.s\ z)\ s\ z) && (6)
\end{aligned}$$

Note that we do not have to perform any further reductions, as we do not compute under λ-abstraction in call-by-value evaluation.

Question 1 (10 points, λ-calculus), Variant 2

Recall Church-encoded numerals:

$$\begin{aligned}
&c_0 := \lambda s.\lambda z.z\\
&c_1 := \lambda s.\lambda z.s\ z\\
&c_2 := \lambda s.\lambda z.s\ (s\ z)\\
&c_3 := \lambda s.\lambda z.s\ (s\ (s\ z))\\
&\dots
\end{aligned}$$

1. **(5 points)** Write down a lambda term that represents the function $f(n) = 3^n$. The answer should be a of the form $f := t$, where t is a pure lambda term (without any references to definitions). In particular, instead of c_1 you should explicitly write $\lambda s.\lambda z.s\ z$.
2. **(5 points)** Write down call-by-value evaluation sequence for $f\ c_1$. The answer consist of a series of pure lambda terms (starting with expanded version of $f\ c_1$). Each step should be a single call-by-value reduction.

Solution

1. First, let's express $f(n)$ using two equations:

$$\begin{aligned}
f(0) &= 1\\
f(n+1) &= f(n) + f(n) + f(n)
\end{aligned}$$

Recall the sum of Church numerals:

$$n + m := \lambda s.\lambda z.n\ s\ (m\ s\ z)$$

Assuming $r = f(n)$, the sum $f(n) + f(n) + f(n)$ can be represented as the term

$$\lambda s.\lambda z.r\ s\ (r\ s\ (r\ s\ z))$$

Now, we can express f as a lambda term:

$$f := \lambda n.n\ (\lambda r.\lambda s.\lambda z.r\ s\ (r\ s\ (r\ s\ z)))\ (\lambda s.\lambda z.s\ z)$$

2.

$$\begin{aligned}
&f\ c_1 && (7)\\
&\equiv (\lambda n.n\ (\lambda r.\lambda s.\lambda z.r\ s\ (r\ s\ (r\ s\ z)))\ (\lambda s.\lambda z.s\ z))\ (\lambda s.\lambda z.s\ z) && (8)\\
&\to (\lambda s.\lambda z.s\ z)\ (\lambda r.\lambda s.\lambda z.r\ s\ (r\ s\ (r\ s\ z)))\ (\lambda s.\lambda z.s\ z) && (9)\\
&\to (\lambda z.(\lambda r.\lambda s.\lambda z.r\ s\ (r\ s\ (r\ s\ z)))\ z)\ (\lambda s.\lambda z.s\ z) && (10)\\
&\to (\lambda r.\lambda s.\lambda z.r\ s\ (r\ s\ (r\ s\ z)))\ (\lambda s.\lambda z.s\ z) && (11)\\
&\to \lambda s.\lambda z.(\lambda s.\lambda z.s\ z)\ s\ ((\lambda s.\lambda z.s\ z)\ s\ ((\lambda s.\lambda z.s\ z)\ s\ z)) && (12)
\end{aligned}$$

Note that we do not have to perform any further reductions, as we do not compute under λ-abstraction in call-by-value evaluation.

Question 2 (10 points, Racket), Variant 1

Using explicit recursion, implement a Racket function `triplets` that splits input list into triplets (lists of size 3, or less if there are not enough elements). You can use the following functions and special forms: `define`, `cond`, `else`, `list`, `cons`, `empty`, `first`, `rest`, `car`, `cdr`, `length`, `reverse`, `append`, `empty?`, `cons?`, `pair?`, `second`, `third`, `quote`, `or`, `and`, `not`, numerical and comparison operators.

```
(triplets '(a b c d e f g h i j k))
; '((a b c) (d e f) (g h i) (j k))
```

Solution

The solution is straightforward, once we realize that if length of the input list is at most 3, we have a base case:

```
(define (triplets lst)
  (cond
    [(empty? lst)
     empty]
    [(<= (length lst) 3)
     (list lst)]
    [else
     (cons
       (list (first lst) (second lst) (third lst))
       (triplets (rest (rest (rest lst)))))]))
```

Question 2 (10 points, Racket), Variant 2

Using explicit recursion, implement a Racket function `quadruplets` that splits input list into quadruplets (lists of size 4, or less if there are not enough elements). You can use the following functions and special forms: `define`, `cond`, `else`, `list`, `cons`, `empty`, `first`, `rest`, `car`, `cdr`, `length`, `reverse`, `append`, `empty?`, `cons?`, `pair?`, `second`, `third`, `quote`, `or`, `and`, `not`, numerical and comparison operators.

```
(quadruplets '(a b c d e f g h i j k))
; '((a b c d) (e f g h) (i j k))
```

Solution

The solution is straightforward, once we realize that if length of the input list is at most 4, we have a base case:

```
(define (quadruplets lst)
  (cond
    [(empty? lst)
     empty]
    [(<= (length lst) 4)
     (list lst)]
    [else
     (cons
       (list (first lst) (second lst) (third lst) (third (rest lst)))
       (quadruplets (rest (rest (rest (rest lst))))))]))
```

Question 3 (15 points, Racket)

Consider the following definition, specifying ratings of some chess grandmasters:

```
(define grandmasters
  '(("Alireza Firouzja"     "France"   .
       (("Classical" . 2804) ("Rapid" . 2656) ("Blitz" . 2810)))
    ("Ding Liren"           "China"    .
       (("Classical" . 2799) ("Rapid" . 2836) ("Blitz" . 2788)))
    ("Fabiano Caruana"      "USA"      .
       (("Classical" . 2792) ("Rapid" . 2770) ("Blitz" . 2803)))
    ("Ian Nepomniachtchi"   "Russia"   .
       (("Classical" . 2782) ("Rapid" . 2798) ("Blitz" . 2792)))))
```

Using higher-order functions (`map`, `ormap`, `andmap`, `filter`, `foldl`) and **without explicit recursion**, implement a function `champion` that takes a list of grandmasters and the mode (classical, rapid, or blitz) and returns the current champion and their country in for the corresponding mode:

```
(champion grandmasters "Classical")
; ("Alireza Firouzja" . "France")

(champion grandmasters "Rapid")
; ("Ding Liren" . "China")

(champion grandmasters "Blitz")
; ("Alireza Firouzja" . "France")
```

Hint: implement a special comparison function `best-of-two` that takes two grandmasters and finds who is the best in a given mode. Then implement a helper function `best` that takes a mode, a list of grandmasters and returns the best grandmaster of all.

Solution

Assuming we have a function `rating-of` that extracts a particular rating for an individual grandmaster and a given mode, it is easy to define `best-of-two`:

```
(define (best-of-two gm1 gm2 mode)
  (cond
    [(> (rating-of gm1 mode) (rating-of gm2 mode))
     gm1]
    [else gm2]))
```

Now, to define `rating-of`, we need to extract the list of mode-ratings pairs from grandmaster entry (which is the third component in the grandmaster triplet), filter out everything except the required mode, extract the pair (the only element in the resulting list), and extract the rating (which is the second component of the pair):

```
(define (rating-of gm mode)
  (cdr (first
    (filter
      (lambda (mode-rating) (= mode (car mode-rating)))
      (cdr (cdr gm))))))
```

With `best-of-two` and `foldl`, we define `best`:

```
(define (best gms mode)
  (foldl best-of-two (first gms) (rest gms)))
```

Note that the task does not have a valid answer for an empty list of grandmasters, so we assume that `gms` is non-empty and use first element as the initial value.

Finally, it is straightforward to define `champion` by repackaging triple into a pair of name and country:

```
(define (champion gms mode)
  (let [(gm (best gms mode))]
    (cons (car gm) (car (cdr gm)))))
```

Question 4 (5 points, Haskell), Variant 1

Implement a function `nth` that finds nth element of a list. You **must not** use any partial functions or imports.

```haskell
nth :: Int -> [a] -> Maybe a

example1 = nth 0 "hello"              -- Just 'h'
example2 = nth 7 [1, 2, 3, 4, 5]      -- Nothing
example3 = nth 1 [[1], [2, 3], [4]] -- Just [2,3]
```

Solution

Straightforward by recursion over the arguments:

```haskell
nth :: Int -> [a] -> Maybe a
nth _ [] = Nothing
nth n (x:xs)
  | n == 0    = Just x
  | otherwise = nth (n - 1) xs
```

Question 4 (5 points, Haskell), Variant 2

Implement a function `penultimate` that finds the second element from the end of a list (if it exists). You **must not** use any partial functions or imports.

```haskell
penultimate :: [a] -> Maybe a

example1 = penultimate "hello"            -- Just 'l'
example2 = penultimate [5]                -- Nothing
example3 = penultimate [[1], [2, 3], [4]] -- Just [2,3]
```

Solution

Straightforward by recursion over the arguments:

```haskell
penultimate :: [a] -> Maybe a
penultimate []        = Nothing
penultimate (_:[])    = Nothing
penultimate (x:_:[]) = Just x
penultimate (_:xs)    = penultimate xs
```

Alternative implementation:

```haskell
penultimate :: [a] -> Maybe a
penultimate [x, _] = Just x
penultimate (_:xs) = penultimate xs
penultimate []     = Nothing
```

And another one:

```haskell
penultimate :: [a] -> Maybe a
penultimate [] = Nothing
penultimate (x:xs) =
  case xs of
    []   -> Nothing
    y:ys ->
      case ys of
        [] -> Just x
        _  -> penultimate xs
```

Question 5 (10 points, Haskell)

Consider the following types:

```haskell
type DoorId = Int

data Item = Key DoorId | Coin

data Tile
  = Wall
  | Floor (Maybe Item)
  | Door DoorId

data Grid a = Grid [[a]]
```

Using higher-order functions (`map`, `filter`, `find`, `concatMap`, `foldl`, `foldr`) and **without explicit recursion**, implement a function `removeCoins` that removes `Coin`s from all `Floor` tiles in a `Grid` of `Tile`s. You **must not** use any partial functions or imports.

```haskell
exampleGrid1 = Grid
  [ [Wall, Floor Nothing, Floor (Just (Key 3))]
  , [Door 3, Floor (Just Coin), Wall] ]

exampleGrid2 = removeCoins exampleGrid1
-- Grid
--  [ [Wall, Floor Nothing, Floor (Just (Key 3))]
--  , [Door 3, Floor Nothing, Wall] ]
```

Solution

Straightforward by structure traversal (using `map`):

```haskell
removeCoins :: Grid Tile -> Grid Tile
removeCoins (Grid rows) = Grid newRows
  where
    -- new grid consists of updated rows
    newRows = map updateRow rows
    -- updating a row means updating each tile in it
    updateRow row = map updateTile row

    -- if we have a floor tile with a coin
    -- then we remove the coin
    updateTile (Floor (Just Coin)) = Floor Nothing
    -- any other tile remains unchanged
    updateTile tile = tile
```

Question 6 (15 points, Haskell)

Implement a polymorphic function `generateRecursively :: (a -> IO [a]) -> a -> IO [a]`. Expression `generateRecursively f x` should repeat apply `f` to `x` and then repeat for the results, until there are not more elements produced. The implementation should apply `f` to all results produced by `f x` before calling `f` recursively. You **must not** use any partial functions or imports.

```
plusMinus :: Int -> IO [Int]
plusMinus n = do
  print n
  return [n-1, n+1]

example1 = generateRecursively plusMinus 3
-- running example1 should print
-- 3
-- 2
-- 4
-- 1
-- 3 (in the exam version there was a typo)
-- 3
-- 5
-- ...
```

Solution

In the example, we start with a single value 3 and repeated recursive calls to `plusMinus` produces the following tree:

$$\begin{tikzcd}
& & 3 \arrow[d, "\texttt{plusMinus}"] & & \\
& & {[2,\ 4]} \arrow[dl] \arrow[dr] & & \\
& {[1,\ 3]} \arrow[dl] \arrow[d] & & {[3,\ 5]} \arrow[d] \arrow[dr] & \\
{[0,\ 2]} & {[2,\ 4]} & & {[2,\ 4]} & {[4,\ 6]}
\end{tikzcd}$$

Function `generateRecursively` traverses this tree breadth-first, producing list `[3, 2, 4, 1, 3, 3, 5, 0, 2, 2, 4, 2, 4, 4, 6, ...]`. Note that the root of the tree is a single value, while for the rest of the tree we have lists in the nodes. This suggests the most important idea for the solution: implement a helper function that starts with a list instead of a single value. Since we need to do a BFS traversal of the tree, we need to do it layer by layer. This means, we require a recursive function that starts with a layer (a list of values) and produces a (program that returns a) new layer:

```
genByLayers :: (a -> IO [a]) -> [a] -> IO [a]
genByLayers _ [] = return []
genByLayers f xs = do
  results <- sequence (map f xs)
  ys <- genByLayers f (concat results)
  return (xs ++ ys)
```

Here it is important to not forget the base case (empty list), and combine the results of the recursive call with the values in the current layer. Note that `results` above is a list of lists (it has type `[[a]]`), that is why we need to use `concat`.

Implementing `generateRecursively` via `genByLayers` is straightforward — we start with a layer that has just one value:

```
generateRecursively :: (a -> IO [a]) -> a -> IO [a]
generateRecursively f x = genByLayers f [x]
```

Question 7 (5 points, Prolog)

Consider the following facts:

```
noun(alice). noun(apple). noun(table).
verb(eats). verb(likes). verb(supports).
adjective(red). adjective(big). adjective(wooden).
```

Write down a predicate `sentence` that is able to generate sentences of the form SUBJECT VERB OBJECT, where both SUBJECT and OBJECT are either a NOUN, or ADJECTIVE NOUN.

```
?- sentence(S).
...
S = [alice, eats, apple]
...
S = [big, table, supports, red, apple]
...
```

Solution

There are at least two approaches. First, we can define the predicate sentence for all possible lengths of the sentence:

```
sentence([N1, V, N2]) :- noun(N1), verb(V), noun(N2).
sentence([A1, N1, V, N2]) :- adjective(A1), noun(N1), verb(V), noun(N2).
sentence([N1, V, A2, N2]) :- noun(N1), verb(V), adjective(A2),  noun(N2).
sentence([A1, N1, V, A2, N2]) :- adjective(A1), noun(N1), verb(V), adjective(A1), noun(N2).
```

Another approach is to follow definition more precisely:

```
sentence(S) :-
  object(Object),
  verb(Verb),
  subject(Subject),
  append(Object, [Verb], X),
  append(X, Subject, S).

object([N]) :- noun(N).
object([A, N]) :- adjective(A), noun(N).

subject(X) :- object(X).
```

Question 8 (15 points, Prolog)

1. **(6 points)** Implement a predicate `distinctMembers/3` that checks whether a given list (of a given size) contains distinct members of another given list.

```
?- distinctMembers(2, X, [a, b, a, c]).
X = [a, b]
X = [a, c]
X = [b, c]

?- distinctMembers(N, X, [a, b, a, c]).
N = 0, X = []
N = 1, X = [a]
N = 1, X = [b]
N = 1, X = [c]
N = 2, X = [a, b]
N = 2, X = [a, c]
N = 2, X = [b, c]
N = 3, X = [a, b, c]
```

2. **(9 points)** Implement a predicate `unifiableMembers/3` that is able to produce sublists of a given list, such that all elements in a sublist can be unified simultaneously:

```
?- unifiableMembers(2, X, [Y, t(Z), t(a), t(b)]).
X = [Y, t(Z)]
X = [Y, t(a)]
X = [Y, t(b)]
X = [t(Z), t(a)]
X = [t(Z), t(b)]

?- unifiableMembers(N, X, [Y, t(Z), t(a), t(b)]), N > 2.
N = 3, X = [Y, t(Z), t(a)]
N = 3, X = [Y, t(Z), t(b)]
```

Solution

1. To implement `distinctMembers/3`, we have to consider three cases:
   (a) base case is when the output list is empty (its size is zero);
   (b) recursive case when we skip the head of a (non-empty) input list; note that we can always skip an element in the input list is we can extract the necessary list from the tail;
   (c) recursive case when we take the head of a (non-empty) input list; here we have to check that head is distinct from the other elements in the output list (produced recursively).

```
distinctMembers(0, [], _).
distinctMembers(N, L, [H|T]) :-
  distinctMembers(N, L, T).
distinctMembers(N, [H|L], [H|T]) :-
  distinctMembers(M, L, T),
  N is M+1,
  \+ member(H, L).
```

2. Implementing `unifiableMembers/3` is similar to `distinctMembers/3`, with the only difference in the last clause: instead of using `member/2` we will be using predicate `unifiableWithAll/2` that checks if a given term can be unified with all terms in the list:

```
unifiableWithAll(X, []).
unifiableWithAll(X, [H|T]) :-
  \+ \+ X = H,
  unifiableWithAll(X, T).

unifiableMembers(0, [], _).
unifiableMembers(N, L, [H|T]) :-
  unifiableMembers(N, L, T).
unifiableMembers(N, [H|L], [H|T]) :-
  unifiableMembers(M, L, T),
  N is M+1,
  unifiableWithAll(H, L).
```

Note that "unifiability" is not a transitive relation, so it would be more correct to use predicate `allUnifiable/1` instead of `unifiableWithAll/2`:

```
allUnifiable(L) :- \+ \+ unifyAll(L).

unifyAll([]).
unifyAll([_]).
unifyAll([X,Y|T]) :-
  X = Y, unifyAll([Y|T]).

unifiableMembers(0, [], _).
unifiableMembers(N, L, [H|T]) :-
  unifiableMembers(N, L, T).
unifiableMembers(N, [H|L], [H|T]) :-
  unifiableMembers(M, L, T),
  N is M+1,
  allUnifiable([H|L]).
```