1. (20pt) Consider the following grammar, G_1, for λ-expressions:

$$\begin{array}{lcl} S & \longrightarrow & E\,\texttt{\$} \\ E & \longrightarrow & \texttt{n} \\ E & \longrightarrow & \texttt{(}\,\lambda\,\texttt{n . }E\,\texttt{)} \\ E & \longrightarrow & \texttt{(}\,E\;E\,\texttt{)} \end{array}$$

   (a) (2pt) Compute the sets FIRST(X) and FOLLOW(X) for all nonterminals $X \in \{S, E\}$.

   (b) (8pt) Show that this grammar is not LL(1) and construct an equivalent one, G_1', that is LL(1). Construct the prediction table to prove that G_1' is LL(1).

   (c) (8pt) Draw the LR-graph for G_1; the states contain LR-items, the transitions are labelled by symbols, and reduce states are double circled. Include also (as jflap does and as shown in class) the trivial states, those containing a single LR-item with the dot at the end.

   (d) (2pt) Is G_1 SLR(1)? Explain briefly your answer.

2. (20pt) Consider the following grammar, G_2, also for λ-expressions:

$$\begin{array}{lcl} S & \longrightarrow & E\,\texttt{\$} \\ E & \longrightarrow & N \\ E & \longrightarrow & \texttt{(}\,\lambda\,N\,\texttt{.}\,E\,\texttt{)} \\ E & \longrightarrow & \texttt{(}\,E\;E\,\texttt{)} \\ N & \longrightarrow & \texttt{a}|\texttt{b}|\cdots|\texttt{z} \end{array}$$

Recall that consecutive abstractions can be written in compressed form, e.g., $\lambda x.\lambda y.e$ becomes $\lambda xy.e$.

(a) (10pt) Write an attributed grammar, based on G_2 above, that, for each λ-expression, produces an equivalent one with all consecutive abstractions compressed, stored in an attribute `str` of S. (Note that the expressions produced by the grammar are fully parenthesized, and they remain fully parenthesized after compressing abstractions, that is, $(\lambda x.(\lambda y.(e)))$ becomes $(\lambda xy.(e))$. There is no restriction on the number or type of attributes used.)

(b) (3pt) Give the compressed version of the following string:

`(λx.(λy.(λz.((x y) z))))$` .

(c) (7pt) Draw the annotated parse tree for the following string, showing the attribute flow:

`(λx.(λy.(x y)))$` .

3. (10pt) The Fibonacci numbers are defined by $F_0 = F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$, for all $n \geq 2$. Consider the following three functions for computing the n^{th} Fibonacci number (the `inexact->exact` procedure in the third function produces an integer as the output):

```
def fibonacci1(n):
    if (n == 0) or (n == 1):
        return 1
    else:
        return fibonacci1(n-1) + fibonacci1(n-2)

def fibonacci2(n):
    (a, b) = (0, 1)
    for i in range(n):
        (a, b) = (b, a+b)
    return b

def fibonacci3(n):
    return int((pow((1 + pow(5, .5))/2, n+1) - pow((1 - pow(5, .5))/2, n+1)) / pow(5, .5))
```

The functions are assumed correct. The first is recursive, the second iterative, and the third uses closed formula:

$$F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n+1} .$$

(a) (5pt) Which of the three functions above is expected to run the slowest? Explain your answer.

(b) (5pt) What implications does your answer to (a) above have on the general competition between the speed of recursive and iterative code? Explain your answer.

4. (15pt) Reduce each of the following λ-expressions to its normal form, using applicative order:
   (a) (5pt) $(\lambda x.x)\,(\lambda x.x)$.
   (b) (5pt) $(\lambda b.a\,(\lambda b.b\,b)\,a)\,(\lambda a.((\lambda b.b)\,a))$.
   (c) (5pt) $(\lambda x.x\,x)\,(\lambda x.x\,x\,x)$.

   Show all computations; indicate each β-reduction by underlying the abstraction and the argument: $(\underline{\lambda x}.M)\underline{N}$.

5. (15pt) Writer a Scheme function, `member-twice`:

```
(define member-twice
  (lambda (x L) ...
```

that returns true if the element `x` occurs in the list `L` at least twice, and false otherwise, where membership is based on the `equal?` predicate. Here is some example behaviour:

```
(member-twice 'a '()) => #f
(member-twice 'a '(a)) => #f
(member-twice 'a '(a a)) => #t
(member-twice 'a '(a b b c)) => #f
(member-twice 'a '(b a c a a)) => #t
```

You are required to provide a purely functional implementation from scratch, that does not employ advanced functions or imperative features. Therefore, you are allowed to use *only* the following basic Scheme functional constructs:
– function creation: `lambda`
– binding: `define`, `let`, `let*`, `letrec`
– booleans: `not`, `and`, `or`
– conditionals: `if`, `cond`
– list operations: `car`, `cdr`, `cons`, `list`, `append`, `null?`
– element comparison: `equal?`

Also, you are not allowed to use additional functions outside the definition of `member-twice`.

6. (20pt) Consider the following Prolog program:

```
subseq([], _).
subseq([H|L], [H|M]) :- subseq(L, M).
subseq(L, [_|M]) :- subseq(L, M).

subseq1([], _).
subseq1([H|L], [H|M]) :- subseq1(L, M).
subseq1([H|L], [X|M]) :- not(H = X), subseq1([H|L], M).
```

Draw the Prolog trees for the computation below; like in the examples we discussed in class, you need to show, on each brach, the rule used and the substitution (if any), the branches not attempted because of cuts (`!`), if applicable (indicate clearly which cut, in the tree, prevented the investigation of which branch), and the output produced (`true` or `false`).

(a) (8pt)

```
?- subseq([a, b], [a, b, a]).
true ;
true ;
false.
```

(b) (12pt)

```
?- subseq1([a, b], [a, b, a]).
true ;
false.
```