1. **Primitive Recursion over nat** (30 Points)

For each of the following three functions give first a specification and then an implementing term. Follow the example of double given in the lecture notes. You may freely reuse the functions from the lecture notes and define your own auxiliary functions.

- **power2 : nat → nat**. For \( n \in \text{nat} \) the term \( \text{power2} \ n \) should compute \( 2^n \).

- **power : nat → nat → nat**. For \( n, m \in \text{nat} \) the application \( \text{power} \ n \ m \) should reduce to \( n^m \).

- **fib : nat → nat**. For \( n \in \text{nat} \) the term \( \text{fib} \ n \) computes the \( n \)th Fibonacci number, where \( \text{fib} \ 0 = 0, \text{fib} \ 1 = 1, \text{fib} \ 2 = 1, \ldots, 2, 3, 5, 8, 13, \ldots \). In this sequence, every number except the first two is the sum of the two preceding numbers.

  Hint: You will need a hack similar to the one in lecture.

2. **Primitive Recursion over list** (30 Points)

Again, give specifications and implementations for the following functions.

- **filter : (τ → bool) → τ list → τ list**. For \( p \in \tau \rightarrow \text{bool} \), \( l \in \tau \text{list} \) the call \( \text{filter} \ p \ l \) returns a sublist \( l' \) of \( l \) which contains only those elements \( x \in \tau \) for which \( p \ x \) returns \true{}.

- **exists : (τ → bool) → τ list → bool**. For \( p \in \tau \rightarrow \text{bool} \), \( l \in \tau \text{list} \) the result of \( \text{exists} \ p \ l \) should be \true{} if \( p \ x \) returns \true{} for any list element \( x \in \tau \), otherwise \false{}.

- **nth : nat → τ list → τ → τ**. For \( n \in \text{nat} \), \( l \in \tau \text{list} \) and \( a \in \tau \) the call \( \text{nth} \ n \ l \ a \) should return the \( n \)th element of the list \( l \), where we start counting in the head with \( 0 \). The value \( a \) should be returned in any exceptional case.

3. **Encoding of bool** (20 Points)

In the lecture the type constructors \( \rightarrow, \times, \text{1} \) and \text{0} were introduced, which are isomorphic to implication, conjunction, truth and falsehood. Here we complete the picture giving the sum type constructor ‘+’ which is isomorphic to disjunction. The rules are:
- Formation: \[
\begin{array}{c}
\sigma \text{ type} & \tau \text{ type} \\
\hline
\sigma + \tau \text{ type} \quad +F
\end{array}
\]

- Introduction:
\[
\begin{array}{c}
\Gamma \vdash t \in \sigma \\
\hline
\Gamma \vdash \text{inl} \, t \in \sigma + \tau \quad +\text{IL}
\end{array}
\quad \begin{array}{c}
\Gamma \vdash t \in \tau \\
\hline
\Gamma \vdash \text{inr} \, t \in \sigma + \tau \quad +\text{IR}
\end{array}
\]

- Elimination:
\[
\begin{array}{c}
\Gamma \vdash r \in \sigma + \tau \\
\Gamma, x \in \sigma \vdash s \in \rho \\
\Gamma, y \in \tau \vdash t \in \rho \\
\hline
\Gamma \vdash \text{case } r \text{ of inl } x \Rightarrow s \mid \text{inr } y \Rightarrow t : \rho \quad +\text{E}
\end{array}
\]

Now we can define a type of booleans as a two-element set: \( \text{Bool} = 1 + 1 \). Convince yourself that the type \( \text{Bool} \) has exactly 2 normal elements and define:

- The truth values \( tt, ff \in \text{Bool} \).
- A term \( \text{ifThenElse} : \text{Bool} \to \tau \to \tau \to \tau \). For \( b \in \text{Bool} \) and \( s, t \in \tau \) the call \( \text{ifThenElse } b \, s \, t \) should return \( s \) if \( b = tt \) and \( t \) otherwise.
- A term \( \text{xor} : \text{Bool} \to \text{Bool} \) that implements exclusive-or.

Again, give specifications and implementations.

4. Binary Trees (20 Points)

In the same manner as natural numbers and lists, we want to introduce labelled complete binary trees as an inductive datatype. Each interior node carries a label and has exactly two child nodes. Each leaf has neither a label nor a child node. Here is an example of a tree carrying natural numbers:

- Give formation, introduction and elimination rules for the data type \( \tau \text{tree} \) which should have the two constructors \( \text{leaf} \) and \( \text{node} \).
- Give a specification and an implementation for each of the functions \( \text{count} : \tau \text{tree} \to \text{nat} \) and \( \text{traverse} : \tau \text{tree} \to \tau \text{list} \). The function \( \text{count} \) counts the number of labels in the given tree (that is 5 in our example) and \( \text{traverse} \) sequentializes all labels into a list such that the leftmost is first and the rightmost is last ([0,1,2,3,4] in our example). Reuse functions defined in the lecture.

Good luck!